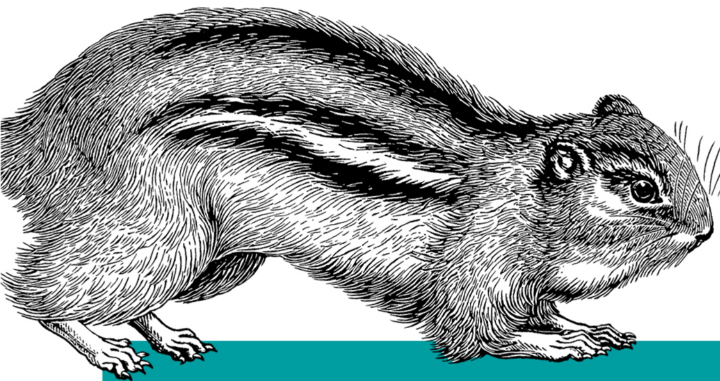
O'REILLY®

TURING 图灵程序设计丛书





HTTP/2 基础教程

Learning HTTP/2: A Practical Guide for Beginners

让Web性能更上一层楼

[美] Stephen Ludin Javier Garza 著

罗正龙 郑维智 译

余晟 审校

中国工信出版集团 人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

译者介绍



罗正龙

沪江前端工程师。从事Web开发超过十年，积累了大量HTTP的经验。目前专注于Node.js相关Web应用开发。



郑维智

沪江（北京研发中心）前端工程师，执着探求Web开发领域的优秀解决方案。

TURING 图灵程序设计丛书

HTTP/2基础教程

Learning HTTP/2
A Practical Guide for Beginners

[美] Stephen Ludin Javier Garza 著
罗正龙 郑维智 译
余晟 审校

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®
O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北 京

图书在版编目（C I P）数据

HTTP/2基础教程 / (美) 斯蒂芬·卢丁
(Stephen Ludin) , (美) 哈维尔·加尔萨
(Javier Garza) 著 ; 罗正龙, 郑维智译. -- 北京 : 人
民邮电出版社, 2018.1
（图灵程序设计丛书）
ISBN 978-7-115-47389-9

Ⅰ. ①H… Ⅱ. ①斯… ②哈… ③罗… ④郑… Ⅲ. ①
计算机网络－通信协议－教材 Ⅳ. ①TN915.04

中国版本图书馆CIP数据核字(2017)第303412号

内 容 提 要

如今互联网发展日新月异，HTTP/1.1 协议已经难以承载日益复杂的网页内容，因此 HTTP/2 值得尝试。本书介绍了 HTTP/2 的设计初衷和新特性，对比了在不同网络环境下以及不同浏览器上 HTTP/1.1 与 HTTP/2 的性能表现差异，指出了网站迁移到 HTTP/2 需要注意的问题，并在附录中给出了书中用到的所有资源的列表，方便读者快速上手实践。

本书适合网站开发及运维人员，以及正考虑要实现 HTTP/2 或者希望了解 HTTP/2 如何工作的读者。

◆ 著　　[美] Stephen Ludin Javier Garza
译　　罗正龙　郑维智
审　校　余　晟
责任编辑　朱　巍
执行编辑　温　雪
责任印制　彭志环

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：8.5
字数：201千字　　2018年1月第1版
印数：1－4 000册　　2018年1月北京第1次印刷

著作权合同登记号　图字：01-2017-8331号

定价：49.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

中文版推荐序一：等待与希望

HTTP/2 已经渗入普通人的生活。如果你平时多个心眼，会发现很多网站已经悄然采用了 HTTP/2，享受了 HTTP/2 带来的诸般好处，甚至在一些技术大会上已经有相关的主题分享。

另一方面，大家对它的了解还相当粗浅。从我主持面试的经历来看，在我认为“应当了解 HTTP/2”的候选人——不管后端还是前端——当中，大部分人还处在“听说过有这么回事”的阶段，只有不到 20% 的候选人能够说出一点实质性的内容。如果继续问“HTTP/2 和 HTTPS 是什么关系”“从 HTTP/1.1 升级到 HTTP/2 有什么要注意的”“HTTP/2 为什么不叫 HTTP/2.0”，能答上来的人就寥寥无几了。

为什么会出现这种情况？我觉得和 HTTP/2 的中文资料匮乏有关。

纵观近年来的中文技术图书市场，无论是涉及的领域，还是作品的质量，都有明显的进步，众星捧月追求英文原版的情况已经是过去时。这对广大技术从业者来说，无疑是好事。但是另一方面，这种情况也间接造成了中英文技术资料的割裂：没有中文图书，大家最多看看网上的文章，没有那么多人愿意去研读英文图书了。关于 HTTP/2，到目前为止，还没有看到过任何中文图书。

难道是因为 HTTP/2 不重要吗？答案显然是否定的。

我最早进入互联网行业时，仅仅满足于“写好程序在网上能跑就行”。随着工作经历的丰富，我越来越深刻地意识到，只要你在这个行业从事技术，无论是前端还是后端，网络相关的基础知识这一课终究是躲不过去的。无论是服务器端要面对的大负载和高并发，还是客户端要面对的有限计算资源和弱网通信环境，最后都离不开对网络的深入理解。

对 TCP/IP 的了解越深，我们往往越会感叹其层次设计的巧妙。各种新出现的优化并不会破坏原有的体系结构。但是，对 HTTP 的了解越深，我们往往越会感叹它的过时——Web 的发展太迅猛了，相比之下，定稿于 1999 年的 HTTP/1.1 时常让人以为是“上古卷轴”。所以，大家才会想出各种“优化黑魔法”来避开 HTTP/1.1 的各种限制和缺陷。域名拆分、

资源域名分离、精灵图（用 CSS 选择大拼图中的小区域）等，都是如此。

不幸的是，这些“高招”往往并非标准统一的解决方案，所以并不能直接放心享用。和 CDN 厂商打过交道就会知道，网络设备成千上万，规范的实现程度也参差不齐，由此产生形形色色的问题，简直让人不胜其烦。再加上各种“优化黑魔法”，只会让本来杂乱的网络世界变得更加混沌。除非自己技术实力足够强，否则只能望洋兴叹。

有没有造福大家的统一的解决办法？ Google 先揭竿而起，发明制订了 SPDY 规范。继而大家才发现，原来所有人都在翘首期盼新的 HTTP 协议。于是，顺理成章地，HTTP/2 诞生了。虽然制订过程是漫长而痛苦的，但首部压缩、分帧传输、服务端推送等新特性，直击 HTTP/1.1 这种“古董协议”的痛点，让广大开发者大呼过瘾，迫不及待想要投入 HTTP/2 的怀抱。

然而，天下没有免费的午餐。要想享受 HTTP/2 带来的诸般好处，“简单升级”协议是行不通的。没错，HTTP/1.1 显得简单直白，相当一部分的开发人员甚至把它理解为一问一答的简单通信模型（和编程语言中的方法调用一样），也不妨碍自己的开发。可惜，HTTP/2 不能这么玩。首部压缩等应用层特性或许还很好懂，但是，新出现的帧传输层绝对要花一番功夫才能理解。不理解这些新特性背后的原理，许多时候就没法调试；不能调试，很多问题就束手无策；束手无策，就无法享受新技术带来的诸般好处。

所幸，《HTTP/2 基础教程》中文版面世了。在我看来，这本书相当适合作为广大开发人员了解 HTTP/2 的资料，大家也确实需要这样一本书，理由如下。

第一，它的范围足够广，HTTP/2 中有价值的新特性基本都有覆盖，不是就概念而概念，而是会讲解各种新特性适合哪些场景以及不适合哪些场景。比如，如果你的网站大量引用第三方资源，那么域名拆分能带来的获益就相当小。

第二，它的编排很用心，不是单纯罗列 HTTP/2 的好处，而是通过与 HTTP/1.1 的对比加以讲解，更有一章专门讲解从 HTTP/1.1 升级到 HTTP/2 的一般过程，以前做的优化哪些必须变更、哪些可以保留，翔实可靠。比如，精灵图这种优化手段依然有助于提升响应速度，但会丧失缓存的便利性。

第三，它不是简单通过定性分析来论证 HTTP/2 的好处，而是大量使用了定量分析的方法。HTTP/2 比 HTTP/1.1 要好，到底好多少，提升的幅度会受哪些因素的影响？书中对这类问题都给出了严谨详细的分析。我读了这本书才知道，光在光纤中的传输速度只有真空中的 2/3。

第四，作为一本优秀的技术图书，其中的分析和思考会让读者在今后的工作中获益更多。本书不仅告诉读者 HTTP/2 的首部压缩采用的是 HPACK 算法，还讲解了为什么采用 HPACK 而不是沿用 SPDY 的 gzip 算法。我相信，如果了解了 CRIME 漏洞的原理，我们在今后的工作中会有更多样的思考角度，以及更完善的安全意识。

《HTTP/2 基础教程》的两位译者罗正龙、郑维智都是沪江优秀的前端开发工程师，本身就对 HTTP 协议有丰富的开发经验，对工作也有高度的责任感，在繁忙的工作之余迅速完成了这本书的翻译。我在审校过程中，经常发现他们就书中的具体问题展开细致的讨论——因为不满足于“翻译文本”，更注重理解和思考背后的原理，所以发现了原书的若干错漏，也确定了很多符合中文开发者习惯的更容易理解的表达方式。我敢说，在今天的中文技术图书译者里，有这样认真精神的人，不超过十分之一。有这样的精神为支撑，图书的翻译质量是有保证的。

写完这篇文章的时候，我再次确认了一番，目前中文技术图书里确实还没有任何一本 HTTP/2 的专著。如果没有估计错误，《HTTP/2 基础教程》中文版的出版只会落后英文原版半年左右，不出意外的话，它应当算中文世界里第一本 HTTP/2 的专著了。我相信，它的质量不会辜负“第一”的名次。

大仲马说过，人类的一切智慧都包含在两个词里——等待与希望。你是不是深深被 HTTP/1.1 所困扰，面对 HTTP/2 又有困惑？那好，现在《HTTP/2 基础教程》终于和读者见面了，让我们共同期待 HTTP/2 的美好未来吧。

——余晟，技术图书翻译写作爱好者，现任沪江教育集团技术中心研发总监

中文版推荐序二

如果有一种远程通信协议堪称“万能协议”的话，那一定非 HTTP 莫属。除了那些对于性能和实时性要求极高的通信场合之外，几乎所有的远程通信都可以基于 HTTP 来实现。Web（全称 World Wide Web）的四大技术基石是 URI、HTML、HTTP 和 MIME，正是这四大基石支撑了宏伟的 Web 神殿。在这四大基石之中，HTTP 的重要性最为突出。今天有很多移动 Web 应用并没有使用 URI、HTML、MIME，只用了 HTTP，仍然可以称为“Web 应用”。从这个角度看，HTTP 几乎是 Web 的代名词。

HTTP 的上一个正式版本是 1.1 版，主设计师是 Roy T. Fielding 博士。HTTP/1.1 与 HTTP/1.0 相比，无论是设计思想还是技术细节方面都取得了巨大的进步。在设计思想方面，Fielding 系统化地提出了 REST 架构风格的理论，以 REST 理论指导 HTTP/1.1 的设计。在 HTTP/1.1 中引入了“资源”这个极为重要的抽象概念，将 HTTP 从一种面向文档的协议彻底转变为一种面向资源的协议。资源是一种非常强大的抽象工具，在 HTTP/1.1 发布之后，Web 之上不再只有大量具体、静态的“文档”，而是包括了无数抽象、动态的“资源”。从对 Web 的概念理解上看，这是一次革命性的转变，推动了 Web 应用的数量以几何级数速度爆发，Web 的范围扩展到了地球上有人类生存的所有地方。可以说，不理解“资源”和相关的“资源表述”，就不理解 HTTP/1.1，对 Web 的理解其实还没有入门。在技术细节方面，HTTP/1.1 也增加了很多新的内容，例如：HTTP 连接支持 keepalive、增加 CONNECT 方法来支持 HTTP tunnel，等等。

这里再强调一下 REST。REST 是 Web 自身的架构风格，也是 Web 取得巨大成功的技术层面的深层原因，理解 REST 就是理解 Web 技术架构的钥匙。前面提到的“资源”及“资源表述”只是 REST 理论之中入门级的概念，REST 还有很多其他重要的概念，例如对于超媒体的有力支持等。建议对 REST 非常感兴趣的读者去读一下 Fielding 的博士论文中文版《架构风格与基于网络应用软件的架构设计》。REST 理论有力地指导了 HTTP/1.1 的设计，也确保了后续 HTTP 协议沿着正确的方向发展，包括从旧版本到新版本的平滑升级。

尽管 HTTP/1.1 取得了辉煌的成就，但 HTTP/1.1 从 1998 年底发布之后，已经十几年都没有更新了，它的很多方面已经难以跟上时代发展的要求。特别是移动互联网普及之后，HTTP/1.1 在性能方面的瓶颈越来越突出，以至于有些公司研发出五花八门的私有二进制 RPC 协议来解决性能问题。这些应用连 HTTP 都不用了，还自称是“Web 应用”，这实在是挂羊头卖狗肉。改进 HTTP 以便跟上新时代 Web 发展的需要，已经迫在眉睫。经过各方参与者几年的不懈努力之后，HTTP/2 终于在 2015 年正式发布了。HTTP/2 是一次非常棒的升级，它在继续遵循 REST 架构风格的前提下，在性能方面取得了巨大的提升。

目前距离 HTTP/2 正式发布已经过去了两年时间，主流的 Web 服务器、浏览器、HTTP 客户端工具、开发库（例如最新的 JDK9）已经能够很好地支持 HTTP/2。可以预见，无论是国内还是国外，2018 年都会是 HTTP/2 迅速普及的一年。国内有很多从事 Web 开发、测试、运维的工程师对 HTTP/2 非常感兴趣，但是苦于缺乏详细的图书文档，难以开展学习。《HTTP/2 基础教程》正是这个领域的最佳图书。这本书虽然不厚，但是满满的都是干货，实战性非常棒。感谢本书的译者罗正龙、郑维智，审校者余晟的辛勤工作，在短时间内将这本高质量的图书贡献给中国的读者。正如本文开头所说的，HTTP 的重要性再怎么强调也不为过。《HTTP/2 基础教程》这本书值得所有软件开发者拥有，作为自己的案头常备图书。

——李锟，Web 架构师，Web 开发老兵

序

截至 2009 年，HTTP/1.1 面世已经超过 10 年了，并且无可争议的是，它依然是互联网上最受欢迎的应用层协议。这是因为它不仅用来浏览网页，还是很多其他东西的参考协议。它上手简单、实现容易，并被广大的开发者和运维工程师所理解，因此积累了很多优势，成为了无可替代的协议。一些人甚至开始说，HTTP 形成了互联网架构经典沙漏模型的“第二腰”。

尽管如此，HTTP 还是与时代脱节了。Web 如今已经发生了翻天覆地的变化，它的需求给 HTTP 协议造成了巨大的压力。现在，加载一个网页通常包含好几百个请求，总体开销在拖慢 Web。这就催生了一个新的行业，专治 Web 性能问题——Web 性能优化。

HTTP 社区很清楚这些问题，但是并没有授权大家修复它们。过往的努力，如 HTTP-NG，已经失败了。没有来自 Web 浏览器和服务器的强力支持的提案就动手的做法，看起来并不妙。这反映在 HTTP 工作组当时的纲领中，它说：

> 工作组不得制定 HTTP 的新版本，也不得给 HTTP 添加新功能。

相反，我们的使命是阐述清楚 HTTP 的规范，并且（至少对我而言）重建一个 HTTP 实现者的强大社区。

也就是说，还有人想实现 HTTP 语义的更高效表达，像 Roy Fielding 的 WAKA 提案[1]（很不幸，从未完成）和基于 SCTP[2] 的 HTTP（主要在特拉华大学）。

去 Google 做了一次有关上面这些话题的分享后，我收到了 Mike Belshe 留给我的一张便条，问我们是否可以碰个面。我们在 Mountain View 的 Castro 大街上吃了晚饭，他说 Google 正要发布 SPDY，替代 HTTP 协议。

注 1：https://tools.ietf.org/agenda/83/slides/slides-83-httpbis-5.pdf

注 2：https://tools.ietf.org/html/draft-natarajan-http-over-sctp-00

SPDY 之所以不同，是因为 Mike 为 Chrome 浏览器工作，他和为 GFE（Google 的前端 Web 服务器）工作的 Roberto Peon 是搭档。他们控制着连接的两端，因而可以快速迭代，并且他们可以在 Google 的超大流量上测试新的协议，因此能够在大规模场景下验证协议的设计。

整个晚饭时间我都感到发自内心的高兴。他们在解决实际问题，并且已经测试过代码和数据。这些正是互联网工程任务组（IETF）所推崇的。

然而，直到 2012 年，SPDY 才开始流行开来。Firefox 实现了该协议，然后是 Nginx 服务器，接着是 Akamai。Netcraft 报告说，支持 SPDY 协议的网站数据激增。

显然，新版本的 HTTP 协议引起了广泛的关注。

2012 年 10 月，HTTP 工作组被授权发布 HTTP/2，使用 SPDY 作为起点。之后的两年间，来自各个公司和开源项目的代表在各地碰面讨论这个新的协议，解决其中的问题，并确保彼此的实现能互相兼容。

在这个过程中，我们有过意见不一致的时候，甚至也有过激烈的争辩。但是，每个人所表现出的专业能力、合作意愿和强烈信念仍然让我印象深刻。这真是一支了不起的、让人愿意共事的团队。

举个例子，有时候大家一致认为，取得进展比为某人的观点争论一整天更重要，所以我们掷硬币来做决定。有些人可能觉得这很疯狂，但我觉得这反映出了成熟的态度和深邃的洞察力。

2014 年 12 月，在规定的截止日期 16 天后（这对标准制定的工作来讲，已经很早了），我们向国际互联网工程指导委员会（IESG）提交了 HTTP/2，申请批准。

大家都说，实践才能出真知；对互联网工程任务组来说，可运行的代码才能说明一切。我们很快就有了拿得出手的东西，并得到了所有主流浏览器、多数 Web 服务器、CDN 和其他工具的支持。

HTTP/2 并不是完美的，但完美向来也不是我们的目标。我们的现实目标是化乱为治，并逐渐提升 Web 性能；远景目标则是为确保**可以**发布新版本的 HTTP 做好准备，这样 Web 才不会受制于一份过时的协议。

从这个角度来看，我们已经成功了。当然，我们要做的还有很多。

——Mark Nottingham

> Mark Nottingham 在 HTTP 工作组已经 10 多年了。对本书来说尤其有意义的是，当 HTTP/2 完成的时候，他担任工作组主席。他现在是工作组负责 QUIC 的主席，也曾是 Akamai Foundry 团队的一员。

前言

HTTP/2，简称 h2，是万维网（World Wide Web）所使用的 HTTP 网络协议的一个重大修订版本，其目的是提升加载 Web 内容时的感知性能。

自从 1999 年 HTTP/1.1（h1）通过以来，Web 发生了翻天覆地的变化。最早大小只有几千字节、包含资源只有个位数、主要基于文本的网页，如今已发展为平均大小超过 2MB[1]、包含资源数平均为 140 的富媒体网站。然而，用来传输 Web 内容的 HTTP 协议这些年并没有什么变化。于是一个新的工种出现了：Web 性能专家，他们精于发掘变通办法，在原有协议上提升网页加载速度。大家对性能的期望也改变了——在 20 世纪 90 年代后期，大家愿意为一个页面等上 7 秒，而技术和市场调研公司 Forrester Research 在 2009 年的一项研究中发现，在线购物者期望单个页面能在 2 秒内完成加载，其中很大一部分用户会放弃加载时间超过 3 秒的页面。近期 Google 的一项研究表明，甚至 400 毫秒（一眨眼的时间）的延迟，都可能降低人们的搜索意愿。

这就是 h2 诞生的原因——该协议可以更好地适应如今的复杂页面，同时又不牺牲速度。HTTP/2 的应用在不断增多，因为越来越多的网站管理员意识到，他们可以借此花很小的成本来提高他们网站的感知性能。

我们每天都在使用 h2——它支撑了若干最流行的网站，如 Facebook、Twitter、Google 和 Wikipedia——但很多人并不了解它。我们的目标是讲解 h2 的原理及其性能优势，这样你就可以充分利用它。

本书目标读者

不管你在工作中扮演什么角色 ，只要你的工作涉及网站生命周期的任意部分，本书就会对你有所帮助。本书的目标读者是网站开发和运维人员，以及那些正考虑要实现 h2 或者希望了解 h2 如何工作的人。

注 1：http://httparchive.org/trends.php?s=Top1000&minlabel=Oct+15+2015&maxlabel=Oct+1+2016

阅读本书要求你熟悉 Web 浏览器、Web 服务器、网站和 HTTP 协议的基础知识。

本书涵盖范围

本书的目标是讲解 h2，并帮你充分利用新版的 HTTP 协议。本书不是一份针对所有 h2 客户端、服务器、调试工具、性能基准测试的全面指南。本书虽然是为不太熟悉 HTTP/2 的人而准备的，但专家没准儿也会觉得这是份称手的资源。

推荐资源

你可以浏览本书在 O'Reilly 网站上的页面（http://shop.oreilly.com/product/0636920052326.do）获取更多信息。此外，我还推荐以下这些书。

- 《高性能网站建设指南》，作者 Steve Souders，前端工程师的基础知识。
- 《高性能网站建设进阶指南》，作者 Steve Souders，Web 开发者的性能最佳实践。
- 《Web 性能权威指南》[2]，作者 Ilya Grigorik，快速上手指南，关于各种网络、传输协议、应用协议，以及浏览器中可用的 API。
- 《WebPageTest 应用指南》，作者 Rick Viscomi、Andy Davies 和 Marcel Duran，介绍 WebPagetest 的基础和高级应用，WebPagetest 是一个用以优化网站的免费性能检测工具。
- *High Performance Mobile Web*[3]，作者 Maximiliano Firtman，介绍如何优化移动网站和移动应用性能。
- *http2 explained*（https://daniel.haxx.se/http2/），作者 Daniel Stenberg。

排版约定

本书使用了下列排版约定。

- **黑体**
  表示新术语或重点强调的内容。

- 等宽字体（`constant width`）
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

- 加粗等宽字体（**`constant width bold`**）
  表示应该由用户输入的命令或其他文本。

注 2：该书已由人民邮电出版社出版，书号：9787115349101。——编者注

注 3：该书中文版即将由人民邮电出版社出版，暂名《高性能移动 Web 开发》，参见 http://www.ituring.com.cn/book/1911。——编者注

- 等宽斜体（*`constant width italic`*）
  表示应该由用户输入的值或根据上下文确定的值替换的文本。

该图标表示提示或建议。

该图标表示一般注记。

该图标表示警告或警示。

使用代码示例

补充材料（代码示例、练习等）可以从 https://github.com/oreillymedia/learning-http2 下载。

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无需联系我们获得许可。比如，用本书的几个代码片段写一个程序就无需获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无需获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如："*Learning HTTP/2* by Stephen Ludin and Javier Garza (O'Reilly). Copyright 2017 Stephen Ludin and Javier Garza, 978-1-491-96244-2."

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

O'Reilly Safari

Safari（前身为 Safari Books Online）是为企业、政府、教育机构和个人提供的会员制的培训和参考平台。

会员可以观看和收听来自 250 多家出版商的上千种图书、培训视频、学习路径、互动教程和推荐歌单。这些出版商包括 O'Reilly Media、Harvard Business Review、Prentice Hall

Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Adobe、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology，等等。

欲知更多信息，请访问 https://www.safaribooksonline.com/。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：http://shop.oreilly.com/product/0636920052326.do。

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com。

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：
http://www.oreilly.com。

我们在 Facebook 的地址如下：http://facebook.com/oreilly。

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia。

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia。

致谢

我们要感谢 Akamai 的 h2 核心团队和 Moritz Steiner（Akamai 的研究员，也是 Foundry 团队的一员，与 Stephen 共同撰写了几篇有关 h2 的论文）；Pierre Lermant（感谢他的幽默和对细节的关注，也感谢他审阅本书并贡献了部分内容）；Martin Flack（也是 Akamai Foundry 团队的一员，时常会提出颇具启发意义的 Lisp 实现）；Jeff Zitomer（感谢他的支持、鼓励，以及富有感染力的微笑）；Mark Nottinghan（感谢他对 h2 协议的贡献）；Pat Meenan（感谢

他为 Webpagetest.org 做出的数不清的贡献，这可能是测试 Web 性能最棒的免费工具了）；Andy Davies（本书中广泛使用的工具 WebPagetest Bulk Tester 就是他开发的）。

感谢本书的编辑 Brian Anderson、Virginia Wilson 和 Dawn Schanafelt，有了他们，一切才变得简单。还要感谢为本书提供反馈和意见的每一位 h2 专家：Ilya Grigorik、Patrick McManus、Daniel Stenberg、Ragnar Lonn、Colin Bendell、Mark Nottingham、Hooman Beheshti、Rob Trace、Tim Kadlec 以及 Pat Meenan。

Javier Garza

首先，我要感谢妻子 Tina 的支持、鼓励和理解。感谢我的孩子们（Keona、Diego 和 Lani），在我花费无数个夜晚和周末以及大段的暑假时间写作本书的时候，他们一如既往地爱着我。感谢我的经理 Aditi 和 Austin，在工作非常紧张的情况下，依然鼓励我写作本书。

Stephen Ludin

我最想感谢的是身边人的耐心。我家人的耐心——感谢 Sarah、Tomas 和 Liam 容忍我以及令人抓狂的出版过程。在写作过程中，他们的支持是无价的。感谢我的雇主 Akamai 允许我在工作非常忙的时候写这本书。感谢 O'Reilly 工作人员的耐心，他们非常理解 O'Reilly 作者平日里的工作已经非常繁忙，只能利用零散的碎片时间来写作。最后，感谢我父母的耐心，他们的付出我无以为报，唯有珍惜当下——在我 9 岁的时候，父亲带回家一台 Atari 800，那时他是否知道，这使我从此走上了一条路，并且如今依然奋力前行。

电子版

扫描如下二维码，即可购买本书电子版。

第 1 章

HTTP进化史

在 20 世纪 30 年代，美国 MIT 工程学院的电子工程师 Vannevar Bush 研究了人们生产的信息量和社会消化信息的能力之间的关系。在 1945 年发表于《大西洋月刊》的文章“As We May Think”中，他写道：

> 从专业角度来看，我们传播和查阅研究结果的方法已经过时，现在已经完全不能满足需要了。如果在写作和阅读学术著作上花费的总时间可以被量化分析，那么两者的比例恐怕非常惊人。

他设想了这样一个系统：聚合起来的知识存储在微缩胶卷上，可以极其方便快速地查阅。他进一步指出，这些信息应当有上下文关联性，就像人类大脑构建的知识链接那样。尽管他设想的 memex（扩展存储器）系统从未实现，其思想却影响了后来者。

如今人们所熟知的术语**超文本**于 1963 年左右被创造出来，于 1965 年首次出现在 Ted Nelson 发表的文章中。Nelson 是一位软件设计师，同时也极具前瞻眼光。他提出“超文本”一词的含义如下：

> 大量的书面材料或图像材料以复杂的方式相互联系，因此不便在纸介质上呈现或展示。其中可能包含内容摘要，或者内容分布的脉络图以及各部分的内在联系；也可能包含审阅过其中内容的学者所添加的注释、补充或脚注。[1]

Nelson 想要建立“文档库”，让信息之间互相关联，永不删除，并且方便所有人获取。20

注 1：T. H. Nelson. “Complex information processing: a file structure for the complex, the changing and the indeterminate.” ACM ’65 Proceedings of the 1965 20th national conference.

世纪 70 年代，基于 Bush 的思想，Nelson 在他的 Xanadu 项目中实现了一个超文本系统的原型。很遗憾，该项目没有完成，但为后来者提供了基础。

1989 年，HTTP 开始进入人们的视野。当时在 CERN（欧洲核子研究组织）的 Tim Berners-Lee 提出了一个新的系统[2]，用以记录粒子加速器（指日后的大型强子对撞机）和 CERN 所做实验产生的信息。他采用了 Nelson 的两个概念：超文本，或者说“以一种无约束的方式联系起来的人类可读信息 ”；超媒体，表示“不限于文本”。在提案中，Tim 提出要搭建“通用系统”，它由服务器和众多机器上的浏览器组成。

1.1 HTTP/0.9和HTTP/1.0

HTTP/0.9 是个相当简单的协议。它只有一个方法（GET），没有首部，其设计目标也无非是获取 HTML（也就是说没有图片，只有文本）。

经过随后几年的发展，HTTP 逐渐流行起来。截至 1995 年，世界上有超过 18 000 台服务器在 80 端口处理 HTTP 请求。此协议在 0.9 版本的基础之上有了长足的发展，并于 1996 年通过 RFC 1945[3] 制定为 HTTP/1.0 规范。

1.0 版本为原有的轻量协议新增了大量内容。0.9 版本的**规范**大概有 1 页，1.0 版本则有 60 页。所以，你可以说它从玩具成长为了工具。它包含了一些我们如今非常熟悉的概念：

- 首部
- 响应码
- 重定向
- 错误
- 条件请求
- 内容编码（压缩）
- 更多的请求方法

……

尽管相对于 0.9 版本来说，HTTP/1.0 是一个巨大的飞跃，但仍存在很多瑕疵，尤其是不能让多个请求共用一个连接，缺少强制的 Host 首部，并且缓存的选择也相当简陋。这三点影响了 Web 可扩展的方式，因此应该在这里强调一下。

1.2 HTTP/1.1

1.0 版本刚刚制定，1.1 版本就接踵而来。截至目前，1.1 版本的协议已经使用了 20 多年

注 2：https://www.w3.org/History/1989/proposal.html

注 3：https://tools.ietf.org/html/rfc1945

了。它修复了之前提到的 1.0 版本的大量问题。因为强制要求客户端提供 Host 首部，所以**虚拟主机托管**成为可能，也就是在一个 IP 上提供多个 Web 服务。当使用新的连接指令时，Web 服务器也不需要在每个响应之后关闭连接。这对于提升性能和效率而言意义重大，因为浏览器再也不用为每个请求重新发起 TCP 连接了。

添加的变更如下：

- 缓存相关首部的扩展
- OPTIONS 方法
- Upgrade 首部
- Range（范围）请求
- 压缩和传输编码（transfer-encoding）
- 管道化（pipelining）

> 管道化这种特性允许客户端一次发送所有的请求。但是有些问题阻碍了管道化的普及，服务器仍然只能按顺序响应请求。具体来说，如果某个请求花了很长时间，那么**队头阻塞**（head of line blocking）会影响其他请求。另外，互联网上的服务器和代理常常没有实现管道化特性（这很糟糕），或者实现得有问题（这更糟糕）。

HTTP/1.1 要归功于 HTTP/1.0 的成功，以及该版本运行那几年所积累的经验。

> **HTTP/1.1 的 RFC**
>
> 互联网工程任务组（IETF）发布的由委员会创建的草案中的协议规范，叫作 RFC（Request for Comments，请求注解）。委员会对所有人开放，前提是你有时间和意愿参与。HTTP/1.1 最早在 RFC 2068 中定义，之后被 RFC 2616 取代，最终在 RFC 7230 到 RFC 7235 中增补和修订。

1.3　1.1版本之后

自 1999 年起，RFC 2616（HTTP/1.1 规范）定义了建立现代 Web 所依赖的标准。正式定稿之后，它就不再变化，也不再改进。然而，Web 和我们使用 Web 的方式一直在变，这种变化甚至超出了它的创造者的想象。普通商业网站的交互性和实用性，已经远远超出了当年互相联系的文档库的设想，并且从根本上改变了我们与世界交互的方式。虽然我们今天看到了协议的各种限制，但进化并没有因此停止。

我们能指出的最明显的变化是网页的构成。HTTP 档案库（HTTP Archives）最早的记录是 2010 年，尽管距今的时间相对较短，变化也足够让人震惊。HTTP 协议设计之初的考虑是单次请求获取单个对象，但如今，新加入的每种元素都增加了复杂度，也带来了压力。

1.4 SPDY

2009 年，Google 的工程师 Mike Belshe 和 Roberto Peon 提出了一种 HTTP 的替代方案：SPDY[4]（发音同 speedy）。SPDY 不是第一个希望替代 HTTP 的方案，但它是其中最重要的一个，因为它带来了显而易见的性能提升。在 SPDY 之前，人们普遍认为在商业应用中没有必要对 HTTP/1.1 做出突破性的、不兼容的改变。要兼容浏览器、服务器、网络代理和其他各种各样的中间件，代价极其高昂。

然而，SPDY 改变了这一切。它很快证明了人们想要更高效的协议，并且愿意改变。SPDY 为 HTTP/2 奠定了基础，并证明了其中一些关键特性的合理性，如多路复用、帧和首部压缩等。即使在互联网时代，SPDY 的发展也算"快"的——它很快被整合进了 Chrome 和 Firefox，并最终几乎被所有主流浏览器所采有。而且几乎在同一时间，服务器和网络代理也对 SPDY 提供了必要的支持。

1.5 HTTP/2

2012 年初，HTTP 工作组（IETF 工作组中负责 HTTP 规范的小组）启动了开发下一个 HTTP 版本的工作。其纲领的关键部分 [5] 阐述了工作组对新协议的一些期望。

> HTTP/2.0 被寄予了如下期望：
>
> - 相比于使用 TCP 的 HTTP/1.1，最终用户可感知的多数延迟都有能够量化的显著改善；
> - 解决 HTTP 中的队头阻塞问题；
> - 并行的实现机制不依赖与服务器建立多个连接，从而提升 TCP 连接的利用率，特别是在拥塞控制方面；
> - 保留 HTTP/1.1 的语义，可以利用已有的文档资源（如上所述），包括（但不限于）HTTP 方法、状态码、URI 和首部字段；
> - 明确定义 HTTP/2.0 和 HTTP/1.*x* 交互的方法，特别是通过中介时的方法（双向）；
> - 明确指出它们可以被合理使用的新的扩展点和策略。

工作组发出了征求建议书的通知，并最终决定使用 SPDY 作为 HTTP/2.0 的起点。最终，RFC 7540 在 2015 年 5 月 14 日发布了，HTTP/2 成为正式协议。

下面，本书来为你讲完这个故事。

注 4：http://dev.chromium.org/spdy/spdy-protocol/spdy-protocol-draft1

注 5：https://datatracker.ietf.org/wg/httpbis/charter/

第 2 章

HTTP/2 快速入门

面对新事物时，我们一般不愿意先花费几小时仔细阅读操作手册、使用说明、维护指南、安全建议等。我们只想快速打开包装盒、接上电源、打开开关，然后就能安心享用。对 HTTP/2（h2）来说也是这样。

那么，现在就开始吧。

2.1　启动并运行

其实，你很可能已经每天都在体验 HTTP/2 了。打开一个现代浏览器（比如 Edge、Safari、Firefox 或 Chrome），访问某个常用网站，比如 Facebook、Instagram 或者 Twitter。瞧，你已经在使用 h2 了。老实说，这不怎么让人兴奋，也不是你阅读本书的原因。所以下面我们自己动手搭建一个 h2 服务器，嗯，下一个使用 h2 的大型网站可能就是**你**创建的。

要运行 h2 服务器，主要分两步：

- 获取并安装一个支持 h2 的 Web 服务器；
- 下载并安装一张 TLS 证书，让浏览器和服务器通过 h2 连接。

这两步都不简单，但我们会尽量做到简单。读完本章之后，你就可以跑自己的支持 h2 的服务器。7.4 节将针对这一主题做更详细的解释。

2.2 获取证书

使用证书这个话题本身就可以写本书了。本章将跳过所有的理论部分，让你尽快拿到一张证书来做实验。接下来将探讨三种方法：使用在线资源；自己创建一张证书；从数字证书认证机构（CA）申请一张证书——我们将使用 Let’s Encrypt。需要说明的是，前两个方法将创建一张所谓的自签名（self-signed）证书，它仅用于测试。由于自签名证书不是由 CA 签发的，浏览器会报警。

2.2.1 使用在线证书生成器

有很多在线服务可以生成自签名的证书。因为你不会在自己的安全环境里生成私钥（private key），所以这些证书永远不能用于除了我们将要做的实验之外的任何情况。只需上网搜一搜，你就能找到很多这样的资源，比如 https://www.sslchecker.com/csr/self_signed。

使用这个工具，将生成的证书和密钥分别保存到两个本地文件中，并分别命名为 privkey.pem 和 cert.pem。

2.2.2 自签名证书

`openssl` 工具是应用广泛且容易获得的，可以在 https://www.openssl.org 找到。几乎每个主流平台都有 `openssl` 的对应版本，我们将使用它演示如何生成自签名的证书和私钥。如果你有 Unix/Linux 或者 macOS 操作系统的设备，你可能已经有了这个工具。打开终端，输入以下命令：

```
$ openssl genrsa -out key.pem 2048
$ openssl req -new -x509 -sha256 -key privkey.pem -out cert.pem -days 365 \
    -subj "/CN=fake.example.org"
```

这样你就有了新的私钥 `privkey.pem` 和新的证书 `cert.pem`。

2.2.3 Let’s Encrypt

Let’s Encrypt 是 CA 领域的新玩家，在 2015 年秋天发布了它的 beta 版本。它的目标是让所有人都能容易、自动、免费地获得 TLS 证书。这是 TLS Everywhere（TLS 无处不在）行动的核心思想，简单来说，它的信念是所有的 Web 通信都应该经过加密和鉴权。在这里，我们看中的是“容易”的特性，所以可以立即动手。

尽管还可以选择众多其他客户端和库 [1] 配合 Let’s Encrypt，但电子前哨基金会（EFF）还是推荐使用 Certbot[2] 作为 Let’s Encrypt 的客户端。Certbot 的目标是使证书的获取和维护变成完全

注 1：https://community.letsencrypt.org/t/list-of-client-implementations/2103

注 2：https://certbot.eff.org/

不需要手动干预的过程，它提供从证书获取到证书安装再到 Web 服务器的一条龙服务。

> 想要从 Let’s Encrypt 获取一张证书，你可能需要验证你的域名。这意味着你拥有这个域名，并且可以修改 DNS 或者 Web 服务器来证明。如果你手里没有域名，或者不想这么麻烦，就使用之前所说的自签名的证书好了。

下载操作系统对应的 Cerbot 的说明在下面。本章是为了让大家尽可能容易地体验，所以你不需要关心选哪个 Web 服务器。对于 Linux 类的操作系统，通常最简单的方式是，在运行 Web 服务器的设备上执行下面的命令：

```
$ wget https://dl.eff.org/certbot-auto
$ chmod a+x certbot-auto
```

下载完成之后，像这样执行 `certbot-auto`：

```
$ ./certbot-auto certonly --webroot -w <your web root> -d <your domain>
```

请把命令中的参数设置为你自己的 Web 服务器的文件目录和域名。上面的命令会自动安装所有用到的包，展示一些需要你输入的问题，最后如果都没问题的话，从 Let’s Encrypt 获得证书。你新制作的证书和私钥会被放到 /etc/letsencrypt/live/`<your domain>`：

文件	描述
/etc/letsencrypt/live/<your domain>/privkey.pem	你的证书的私钥
/etc/letsencrypt/live/<your domain>/cert.pem	你的新证书
/etc/letsencrypt/live/<your domain>/chain.pem	Let’s Encrypt 的 CA 证书链
/etc/letsencrypt/live/<your domain>/fullchain.pem	包含证书和证书链的文件

2.3 获取并运行你的第一个HTTP/2服务器

有很多能支持 HTTP/2 的服务器可供选择。（7.4 节会讲几个这样的服务器。）现在我们的目标是尽可能快和简单，因此我们将使用 nghttp2[3] 这个包。nghttp2 由 Tatsuhiro Tsujikawa 开发，提供了很多处理和调试 h2 的称手工具。不过现在，我们只关注 `nghttpd` 这个工具。

关于 nghttp2，8.4 节会有更详细的介绍，但我们希望你现在就直接开始运行。所以，请使用你喜欢的包管理工具安装 nghttp2，或者从源码编译（如果你有勇气的话）。例如，在 Ubuntu 16 上面：

```
$ sudo apt-get install nghttp2
```

安装之后，现在手上已经有了证书，启动 `nghttpd` 如下：

注 3：https://nghttp2.org/

```
$ ./nghttpd -v -d <webroot> <port> <key> <cert>
```

其中 <webroot> 是你网站的路径，<port> 是服务器要监听的端口号，<key> 和 <cert> 是你生成的私钥和证书路径。例如：

```
$ ./nghttpd -v -d /usr/local/www 8443 \
            /etc/letsencrypt/live/yoursite.com/privkey.pem \
            /etc/letsencrypt/live/yoursite.com/cert.pem
```

2.4　选择浏览器

最终，到了检验劳动成果的时候。启动一个现代浏览器，并用它访问你新安装的服务器。7.1 节有支持 HTTP/2 的浏览器的完整列表。如果创建的是一张自签名的证书，你会看到安全警告。请确认浏览器提示的是你创建证书的问题，然后点击接受警告。现在你应该可以看到你的网站了。

这就是基于 h2 的服务！

第 3 章

Web优化“黑魔法”的动机与方式

用相对古老的协议承载响应迅速的现代网页，实际上已经成了某种高超的杂技，并且以其为中心，已经发展出了一整套 Web 性能的技巧。也许有人会说，O’Reilly 举办的 Velocity 系列研讨会，部分缘于这些人的动机——他们希望分享自己的技巧和诀窍，从德高望重的协议上尽可能榨取收益。为了弄明白我们正走向何方（名义上，朝着 HTTP/2），搞清楚现在身处何处、面临什么挑战，以及当前我们如何应对这些问题，这些都是非常重要的。

3.1 当前的性能挑战

发布现代 Web 页面或 Web 应用绝不是一件小事。每个页面会引用数百个对象，关联十多个域名，网络环境相差迥异，设备的处理能力也参差不齐。在这样的前提下，提供表现一致且响应迅速的 Web 体验并不简单。如果要保证用户在网站上的交互流畅，就必须重视理解客户端获取 Web 页面和渲染的步骤，以及各个步骤要面临的挑战。理解了它们，你就能理解推动 HTTP/2 的力量，也能理解其新特性相对 HTTP/1 都带来哪些益处。

3.1.1 剖析Web页面请求

在进一步深入探讨之前，必须先搞清楚我们正在尽力优化的是什么；尤其要搞清楚的是，从用户在浏览器中点击链接到页面呈现在屏幕上，在此期间到底发生了什么？浏览器请求 Web 页面时，会执行重复流程，获取在屏幕上绘制页面需要的所有信息。为了更容易理解，我们把这一过程分成两部分：资源获取、页面解析 / 渲染。先从资源获取的部分开始。图 3-1 展示了它的步骤。

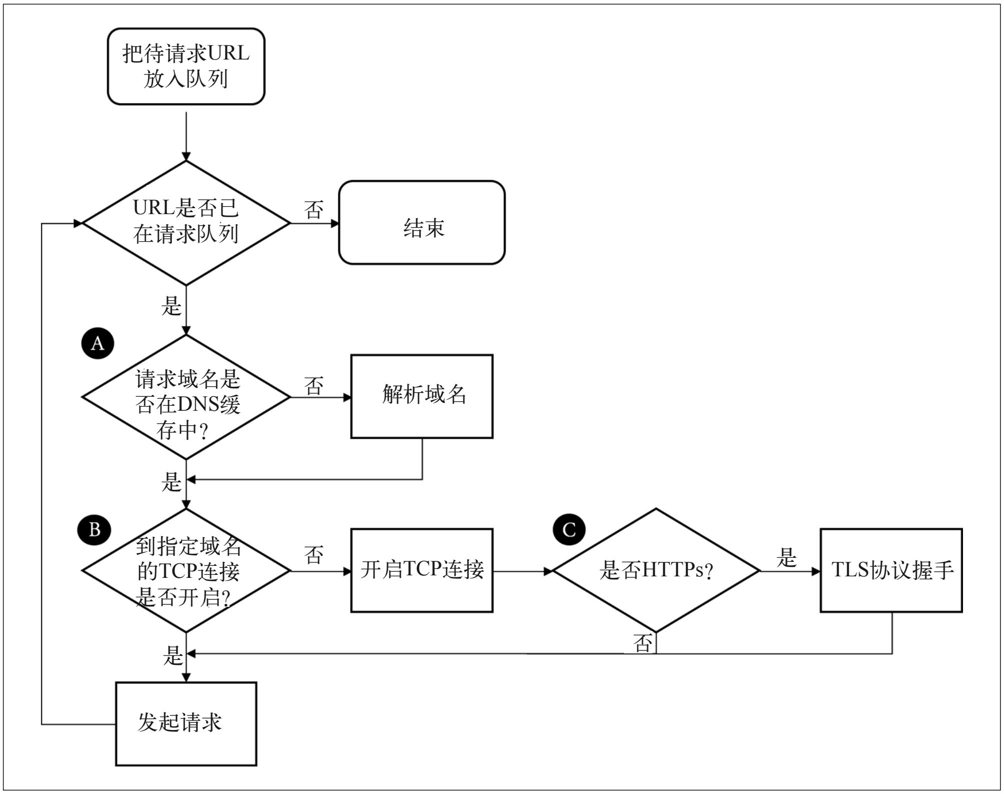


图 3-1：资源请求 / 获取流程图

按流程图走一遍：

(1) 把待请求 URL 放入队列；
(2) 解析 URL 中域名的 IP 地址（A）；
(3) 建立与目标主机的 TCP 连接（B）；
(4) 如果是 HTTPS 请求，初始化并完成 TLS 握手（C）；
(5) 向页面对应的 URL 发送请求。

图 3-2 展示了接收响应及页面渲染的过程。接着走这个流程：

(6) 接收响应；
(7) 如果（接收的）是主体 HTML，那么解析它，并针对页面中的资源触发优先获取机制（A）；
(8) 如果页面上的关键资源已经接收到，就开始渲染页面（B）；
(9) 接收其他资源，继续解析渲染，直到结束（C）。

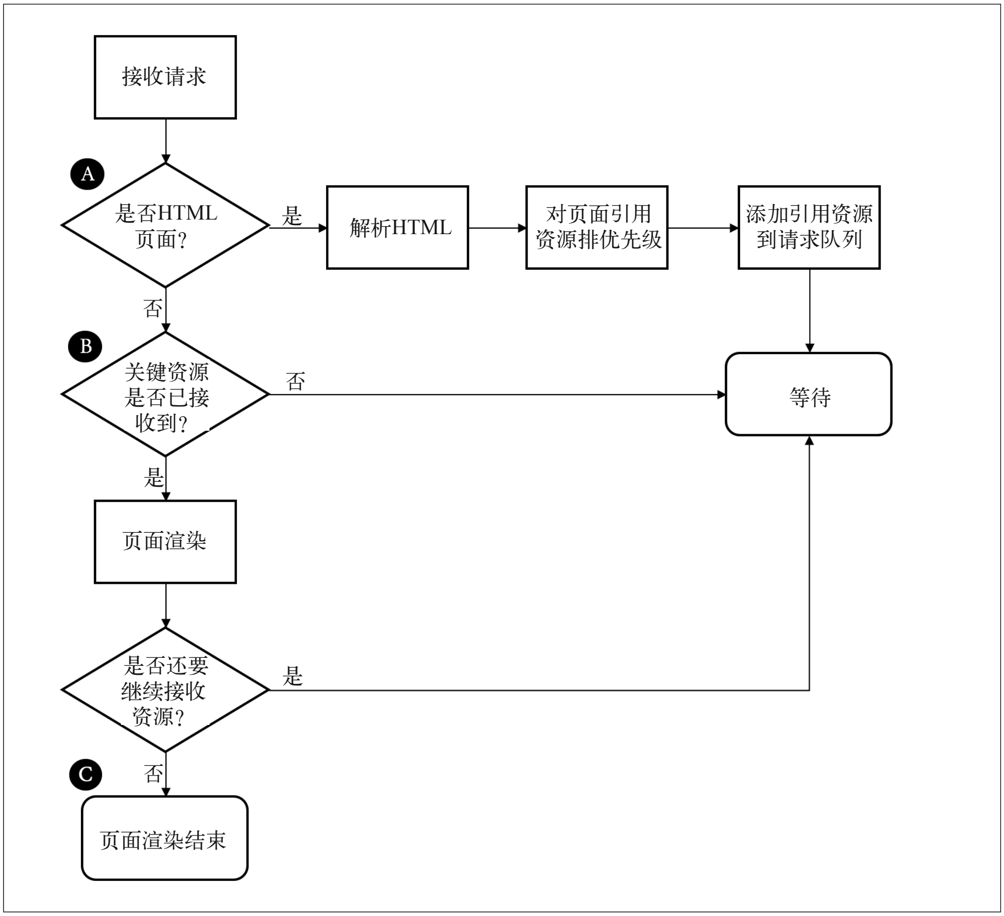


图 3-2：资源响应 / 页面渲染流程图

页面上的每一次点击，都需要重复执行前面那些流程，给网络带宽和设备资源带来压力。Web 性能优化的的核心，就是加快甚至干脆去掉其中的某些步骤。

3.1.2　关键性能指标

从上面的图中，我们能找到影响 Web 性能的端点，以及能动手改进的地方。让我们从网络级别的性能指标开始，它会影响整个 Web 页面加载。

延迟

延迟是指 IP 数据包从一个网络端点到另一个网络端点所花费的时间。与之相关的是往返时延（RTT），它是延迟的时间的两倍。延迟是制约 Web 性能的主要瓶颈，尤其对于 HTTP 这样的协议，因为其中包含大量往返于服务器的请求。

带宽

只要带宽没有饱和，两个网络端点之间的连接会一次处理尽可能多的数据量。依据 Web 页面引用资源的大小和网络连接的传输能力，带宽可能会成为性能的瓶颈。

DNS查询

在客户端能够获取 Web 页面前，它需要通过域名系统（DNS）把主机名称转换成 IP 地址，DNS 相当于互联网上的电话号码簿（今天可能没有多少年轻人知道号码簿了）。获取的 HTML 页面中所引用的各个不同域名也需要转换；幸运的是，一个域名只需转换一次。

建立连接时间

在客户端和服务器之间建立连接需要往返数据应答，称为“三次握手”。握手时间一般与客户端和服务器之间的延迟有关。握手过程包括客户端向服务器发起一个 SYN 包，接着服务器返回对应 SYN 的 ACK 响应以及新的 SYN 包，然后客户端返回对应的 ACK，如图 3-3 所示。

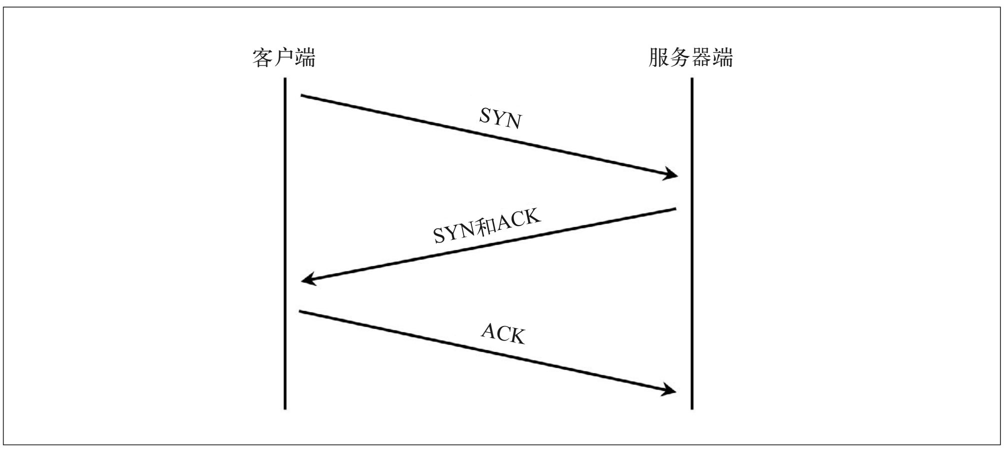


图 3-3：TCP 三次握手

TLS协商时间

如果客户端发起 HTTPS 连接，它还需要进行传输层安全协议（TLS）协商；TLS 用来取代安全套接层（SSL）。除了服务器和客户端的计算处理耗时之外，TLS 还会造成额外的往返传输。

目前为止，客户端还没有真正发起 HTTP 请求，却已经用掉了 DNS 查询的往返时间，以及 TCP 和 TLS 的耗时。下面的指标严重依赖于页面内容本身或服务器性能，而不是网络。

首字节时间（TTFB）

TTFB 是指客户端从开始定位到 Web 页面，至接收到主体页面响应的第一字节所耗费

的时间。它包含了之前提到的各种耗时，还要加上服务器处理时间。对于主体页面上的资源，TTFB 测量的是从浏览器发起请求至收到其第一字节之间的耗时。

内容下载时间

等同于被请求资源的最后字节到达时间（TTLB）。

开始渲染时间

客户端的屏幕上什么时候开始显示内容？这个指标测量的是用户看到空白页面的时长。

文档加载完成时间（又叫页面加载时间）

这是客户端浏览器认为页面加载**完毕**的时间。

如果我们关注 Web 性能，尤其是要制定新协议来提升效率，就必须把（上面提到的）那些指标牢记于心。后文讨论 HTTP/1.1 面临的问题以及我们想要寻求改变的原因时，还会提到它们。

把那些性能指标牢记于心之后，你就能看到互联网的发展趋势：各种东西都在**增加**，所以出现了性能瓶颈。下面是我们需要记住的一些**增加**的方面。

更多的字节

毫无疑问的是，Web 页面引用的内容每年都在增长，图片越来越大，JavaScript 和 CSS 也越来越大。内容体量变大意味着（客户端需要）下载更多的字节，也意味着更长的**页面加载时间**。

更多的资源

页面引用的资源不仅变大，而且数量也增多了。引用更多的资源会导致总耗时增加，因为所有的资源都需要获取并解析。

更高的复杂度

随着我们添加更多、更丰富的功能，Web 页面和所依赖的资源正变得越来越复杂。复杂度提升，伴随而来的是计算渲染 Web 页面的时间不断延长，尤其是在处理能力较弱的移动设备上。

更多的域名

Web 页面并不是从单一的域名拉取下来的，大多数 Web 页面会关联数十个域名。每出现一个新域名都会增加 **DNS 查询耗时**、**建立连接耗时**，以及 **TLS 协商耗时**。

更多的TCP socket

为了应对某些方面的**增加**，客户端会对同一个域名开启多个 socket。这增加了与域名对应的服务器协商建立连接的开销，也加重了设备负担，还有可能导致网络连接过载，引发出错重传和缓存过满，并降低有效带宽。

3.1.3 HTTP/1的问题

HTTP/1 已经支撑我们走到今天，但是现代 Web 应用的需求迫使我们关注其设计缺陷。下面是它面临的一些比较重要的问题；这自然也是设计 HTTP/2 要解决的核心问题。

> 并没有“HTTP/1”这种专业术语；此处这一用法（还有 h1），是对 HTTP/1.0（RFC 1945）和 HTTP/1.1（RFC 2616）的简称。

1. 队头阻塞

浏览器很少只从一个域名获取一份资源。大多数时候，它希望能同时获取许多资源。设想这样一个网站，它把所有图片放在单个特定域名下。HTTP/1 并未提供机制来同时请求这些资源。如果仅仅使用一个连接，它需要发起请求、等待响应，之后才能发起下一个请求。h1 有个特性叫管道化（pipelining），允许一次发送一组请求，但是只能按照发送顺序依次接收响应。而且，管道化备受互操作性和部署的各种问题的困扰，基本没有实用价值。

在请求应答过程中，如果出现任何状况，剩下所有的工作都会被阻塞在那次请求应答之后。这就是“队头阻塞”，它会阻碍网络传输和 Web 页面渲染，直至失去响应。为了防止这种问题，现代浏览器会针对单个域名开启 6 个连接，通过各个连接分别发送请求。它实现了某种程度上的并行，但是每个连接仍会受到“队头阻塞”的影响。另外，这也没有高效利用有限的设备资源；下一节会解释原因。

2. 低效的TCP利用

传输控制协议（TCP）的设计思路是：对假设情况很保守，并能够公平对待同一网络的不同流量的应用。它的避免拥塞机制被设计成即使在最差的网络状况下仍能起作用，并且如果有需求冲突也保证相对公平。这是它取得成功的原因之一。它的成功并不是因为传输数据最快，而是因为它是最可靠的协议之一，涉及的核心概念就是**拥塞窗口**（congestion window）。拥塞窗口是指，在接收方确认数据包之前，发送方可以发出的 TCP 包的数量。例如，如果拥塞窗口指定为 1，那么发送方发出 1 个数据包之后，只有接收方确认了那个包，才能发送下一个。

> **什么是数据包**
>
> 数据包，或更具体来说，是 IP 数据包，指封装在固定结构（例如帧）的一系列字节（或负载），它定义了数据包长度、传输的细节（包的发送者和目的地），以及其他与 TCP 相关的信息。（如果）想有效利用数据包负载的话，单个数据包里面最多可以放 1460 字节。想象一下，一张大小为 14 600 字节（该数字只是为便于举例而设）的图片会如何？它将被拆分成 10 个数据包。一旦理解了数据包（和本节讲述的其他内容），你就能揭开网络性能的面纱。

一般来讲，每次发送一个数据包并不是非常低效。TCP 有个概念叫**慢启动**（Slow Start），它用来探索当前连接对应拥塞窗口的合适大小。慢启动的设计目标是为了让新连接搞清楚当前网络状况，避免给已经拥堵的网络继续添乱。它允许发送者在收到每个确认回复后额外发送 1 个未确认包。这意味着新连接在收到 1 个确认回复之后，可以发送 2 个数据包；在收到 2 个确认回复之后，可以发 4 个；以此类推。这种几何级数增长很快就会到达协议规定的发包数上限，这时候连接将进入**拥塞避免阶段**，如图 3-4 所示。

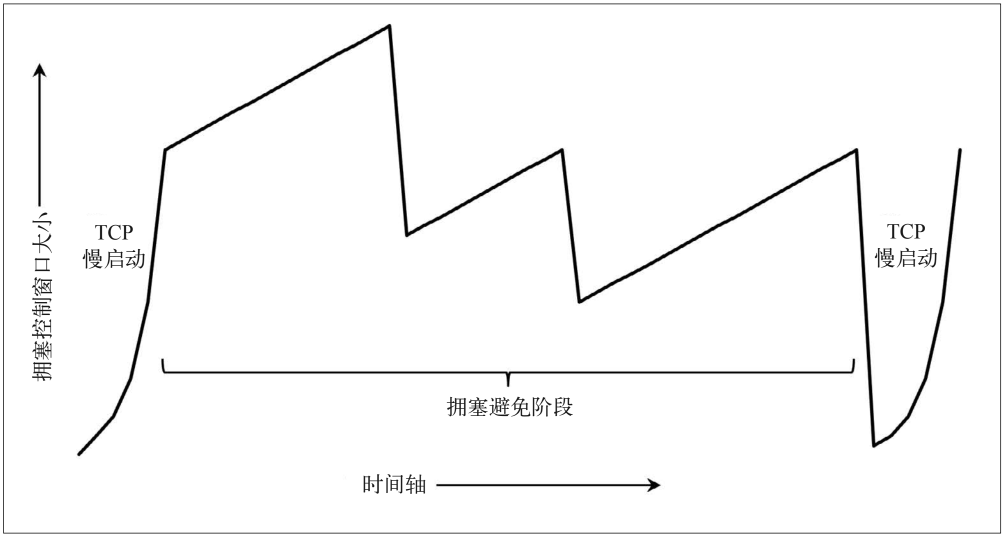


图 3-4：TCP 拥塞控制（Reno 算法）

这种机制需要几次往返数据请求才能得知最佳拥塞窗口大小。但在解决性能问题时，就这区区几次数据往返也是非常宝贵的时间（成本）。现代操作系统一般会取 4~10 个数据包作为初始拥塞窗口大小。如果你把一个数据包设置为最大值下限 1460 字节（也就是最大有效负载），那么只能先发送 5840 字节（假定拥塞窗口为 4），然后就需要等待接收确认回复。如今的 Web 页面平均大小约 2MB，包括 HTML 和所有依赖的资源。在理想情况下，这需要大约 9 次往返请求来传输完整个页面。除此之外，浏览器一般会针对同一个域名开启 6 个并发连接，每个连接都免不了拥塞窗口调节。

传输数据包计算

上面提到的那些数字是怎么得出来的？这个时候了解一些数学知识很有必要，有助于估算对网络传输的影响，看看到底是增加还是减少了传输字节数。假设拥塞窗口的大小每次往返请求之后会翻一番，每个数据包承载 1460 字节。在理想情况下，呈现出等比数列，如下表所示。

数据往返次数	发送数据包个数	发送最大字节数	发送总字节数
1	4	5840	5840
2	8	11 680	17 520
3	16	23 360	40 880
4	32	46 720	87 600
5	64	93 440	181 040
6	128	186 880	367 920
7	256	373 760	741 680
8	512	747 520	1 489 200
9	1024	1 495 040	2 984 240

直到第 9 次，2MB 数据才能全部发完。不过，这过度简化了现实情况。在窗口大小达到 1024 个数据包时，要么会触发一个叫 ssthresh（**慢启动阈值**）的上限值，要么窗口会自动缩小；不管哪种情况，几何级数增长都会终止。不过，用于粗略估算时，这种简单方法已经够用了。

传统 TCP 实现利用拥塞控制算法会根据数据包的丢失来反馈调整。如果数据包确认丢失了，算法就会缩小拥塞窗口。这就类似于我们在黑暗的房间摸索，如果腿碰到了桌子就会马上换个方向。如果遇到超时，也就是等待的回复没有按时抵达，它甚至会彻底重置拥塞窗口并重新进入慢启动阶段。新的算法会把其他因素也考虑进来，例如延迟，以提供更妥善的反馈机制。

前面提到过，因为 h1 并不支持多路复用，所以浏览器一般会针对指定域名开启 6 个并发连接。这意味着拥塞窗口波动也会并行发生 6 次。TCP 协议保证那些连接都能正常工作，但是不能保证它们的性能是最优的。

3. 臃肿的消息首部

虽然 h1 提供了压缩被请求内容的机制，但是消息首部却无法压缩。消息首部可不能忽略，尽管它比响应资源小很多，但它可能占据请求的绝大部分（有时候可能是全部）。如果算上 cookie，有个几千字节就很正常了。

据 HTTP 历史存档记录，2016 年末，请求首部一般集中在 460 字节左右。对于包含 140 个资源的普通 Web 页面，意味着它在发起的所有请求中大约占 63KB。想想之前关于 TCP 拥塞窗口管理的讨论，发送该页面相关的所有请求可能需要 3~4 轮往返，因此网络延迟的损耗会被迅速放大。此外，上行带宽通常会受到网络限制，尤其是在移动网络环境中，于是拥塞窗口机制根本来不及起作用，导致更多的请求和响应。

消息首部压缩的缺失也容易导致客户端到达带宽上限，对于低带宽或高拥堵的链路尤其如此。“体育馆效应”（Stadium Effect）就是一个经典例子。如果成千上万人同一时间出现在

同一地点（例如重大体育赛事），会迅速耗尽无线蜂窝网络带宽。这时候，如果能压缩请求首部，把请求变得更小，就能够缓解带宽压力，降低系统的总负载。

4. 受限的优先级设置

如果浏览器针对指定域名开启了多个 socket（每个都会受队头阻塞问题的困扰），开始请求资源，这时候浏览器能指定优先级的方式是有限的：要么发起请求，要么不发起。然而 Web 页面上某些资源会比另一些更重要，这必然会加重资源的排队效应。这是因为浏览器为了先请求优先级高的资源，会推迟请求其他资源。但是优先级高的资源获取之后，在处理的过程中，浏览器并不会发起新的资源请求，所以服务器无法利用这段时间发送优先级低的资源，总的页面下载时间因此延长了。还会出现这样的情况：一个高优先级资源被浏览器发现，但是受制于浏览器处理的方式，它被排在了一个正在获取的低优先级资源之后。

5. 第三方资源

虽然第三方资源不是 HTTP/1 特有的问题，但鉴于它日益增长的性能问题，我们也把它列在这里。如今的 Web 页面上请求的很多资源完全独立于站点服务器的控制，我们称这些为第三方资源。现代 Web 页面加载时长中往往有一半消耗在第三方资源上。虽然有很多技巧能把第三方资源对页面性能的影响降到最低，但是很多第三方资源都不在 Web 开发者的控制范围内，所以很可能其中有些资源的性能很差，会延迟甚至阻塞页面渲染。任何关于 Web 性能的讨论，只要没有提到第三方资源引起的问题，都不算完整。（令人扫兴的是，h2 对此也束手无策。）

> **第三方资源的代价是什么**
>
> 第三方资源究竟让页面慢多少？ Akamai 的 Foundry 团队的研究显示，第三方资源的影响非常大，平均累计占到页面整体加载时间的一半[1]。这份报告提出了新的用于跟踪第三方资源影响的指标，称为“3rd Party Trailing Ratio”。它测量的是请求并展现第三方内容对页面渲染时间的影响程度。

3.2 Web性能优化技术

21 世纪初，当时在 Yahoo! 工作的 Steve Souders 和他的团队提议并评估了让 Web 页面在浏览器中加载更快的技术。这份研究引领他撰写了两本影响深远的著作：《高性能网站建设指南》和姊妹篇《高性能网站建设进阶指南》，堪称 Web 性能科学的奠基之作。

从那以后，有更多的研究证实，Web 性能和站点所有者关心的核心指标有密切的关系，例

注 1：https://www.akamai.com/us/en/multimedia/documents/technical-publication/are-3rd-parties-slowing-down-the-mobile-web.pdf

如转化率、用户活跃度或品牌意识。2010 年，谷歌把 Web 性能作为影响页面搜索评分的重要因素之一，性能指标开始在搜索引擎中发挥作用[2]。对于大多数企业而言，Web 存在的重要性与日俱增；因此，理解、评估、优化网站性能显得无比重要。

正如本章开头所提及的，对于很多 Web 页面，浏览器的大块时间并不是用于呈现来自网站的主体内容（通常是 HTML），而是在请求所有资源并渲染页面，如图 3-5 所示[3]。

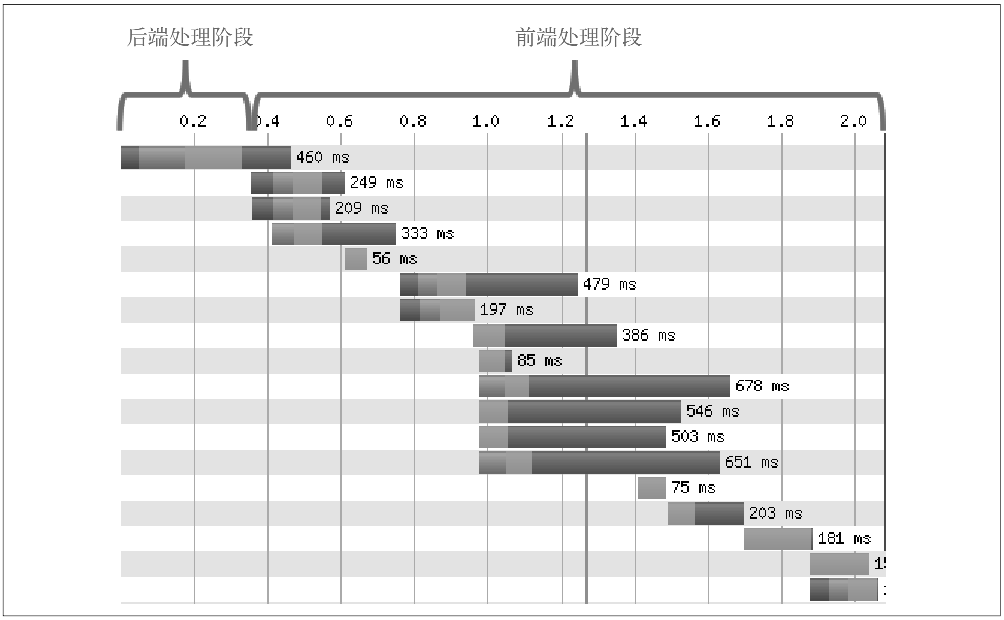


图 3-5：前后端时间轴

因此，Web 开发者逐渐更多地关注通过减少客户端网络延迟和优化页面渲染性能来提升 Web 性能。时间就是金钱，这话讲得一点也不夸张。

3.2.1　Web性能的最佳实践

前面已经讲过，Web 已经发生了很大的变化，即使在过去的短短几年也是如此。而近年来移动设备的普及、JavaScript 框架的发展以及 HTML 使用上的进化，让我们有必要回顾前文引用的两本书中讲述的规则，重温业界最新的优化技术。

1. DNS查询优化

在与服务主机建立连接之前，需要先解析域名；那么，解析越快就越好。可以从下列诀窍开始。

注 2：https://webmasters.googleblog.com/2010/04/using-site-speed-in-web-search-ranking.html

注 3：http://stevesouders.com/images/golden-waterfall.png

- 限制不同域名的数量。当然，这通常不是你能控制的；但是如果准备迁移到 HTTP/2，域名数量对性能的相对影响会只增不减。
- 保证低限度的解析延迟。了解你的 DNS 服务基础设施的结构，然后从你的最终用户分布的所有地域定期监控解析时间（你能通过虚拟或真实用户的监控做到）。如果你要依赖外部供应商，一定要谨慎选择，因为各家的服务质量参差不齐。
- 在主体页面 HTML 或响应中利用 DNS 预取指令[4]。这样，在下载并处理主体页面 HTML 的同时，预取指令就能开始解析页面上指定的域名。例如，下面这段代码会告诉浏览器预解析 ajax.googleapis.com：

```
<link rel="dns-prefetch" href="//ajax.googleapis.com>
```

这些诀窍能帮助确保域名解析的固定开销最小化。

2. 优化TCP连接

本章前面提到过，开启新连接是一个耗时的过程。如果连接使用 TLS（也确实应该这么做），开销会更高。降低这种开销的方法如下。

- 利用 `preconnect` 指令[5]，连接在使用之前就已经建立好了，这样处理流程的关键路径上就不必考虑连接时间了。例如：

```
<link rel="preconnect" href="//fonts.example.com" crossorigin>
```

- 尽早终止并响应。借助 CDN，在距离请求用户很近的边缘端点上，请求就可以获得响应，所以可以终止连接，大幅减少建立新连接的通信延迟。更多信息参见 7.5 节的内容。
- 实施最新的 TLS 最佳实践[6]来优化 HTTPS。

如果要从同一个域名请求大量资源，浏览器将自动开启到服务器的并发连接，避免资源获取瓶颈。虽然现在大部分浏览器支持 6 个或更多的并发连接数目，但你不能直接控制浏览器针对同一域名的并发连接数。

3. 避免重定向

重定向通常触发与额外域名建立连接。在无线网络中（想想手机用户），一次额外的重定向可能把延迟增加数百毫秒，这不利于用户体验，并最终会影响到网站上的业务。简单的解决方案就是彻底消灭重定向，因为对于重定向的使用往往并没有合理原因。如果它们不能被直接消灭，你还有两个选择：

- 利用 CDN 代替客户端在云端实现重定向；
- 如果是同一域名的重定向，使用 Web 服务器上的 rewrite 规则，避免重定向。

注 4：https://www.w3.org/TR/resource-hints/#dns-prefetch
注 5：https://www.w3.org/TR/resource-hints/#preconnect
注 6：https://istlsfastyet.com/

通常，重定向和搜索引擎优化（SEO）的黑魔法一起，用于帮助短期内优化搜索结果，或避免后端信息架构为 SEO 而调整。在这些情况下，你不得不衡量重定向的代价是否抵得上 SEO 的好处。有时候，一次到位地消灭重定向是最好的长期解决方案。

4. 客户端缓存

没有什么比直接从本地缓存获取资源来得更快，因为它根本就不需要建立网络连接。俗话说（或者至少从现在开始说），最快的请求是根本不发起请求。另外，从本地获取资源时，ISP 或 CDN 提供商不会收取流量费。生存时间（TTL）指令告诉浏览器应该缓存某个资源多久。找到给定资源的最佳 TTL 值并没有完美的科学方法。不过，你可以从下面这些已经验证过的指导原则开始。

- 所谓的纯静态内容，例如图片或带版本的数据，可以在客户端永久缓存。尽管如此，我们也要记住，即便 TTL 被设置得很长，比如一个月，它还是会因为缓存提早回收或清理而过期，这时客户端可能不得不从源头再次获取。因此真实的 TTL（效果）最终取决于设备特性（尤其是可用磁盘缓存空间）和最终用户的浏览习惯 / 历史记录。
- CSS/JS 和个性化资源,缓存时间大约是会话（交互）平均时间的两倍。这段时间足够长，保证大多数用户在浏览网站时能够从本地拉取资源；同时也足够短，几乎能保证下次会话时从网络上拉取最新内容。
- 其他类型的资源，理想的 TTL 值会各有不同；这取决于你对特定资源能够容忍的旧数据的极限。所以，你必须结合自身需求来判断最佳值。

设置客户端缓存 TTL，可以通过 HTTP 首部指定 cache control 以及键 max-age（以秒为单位），或者 expires 首部。

5. 网络边缘的缓存

因为所有用户都能从云端的共享缓存受益，所以网络边缘的缓存提供了更快的访问速度，也为网站服务基础设施分担了很大一部分流量。

如果一份资源需要缓存，它必须满足：

- 在多用户间可共享，并且
- 能够接受一定程度的旧数据

个人信息（用户偏好、财务数据等）绝对不能在网络边缘缓存，因为它们不能共享。类似地，时间敏感的资源也不应该缓存，例如实时交易系统上的股票报价。这就是说，（除此之外的）其他一切都是可以缓存的，即使仅仅缓存几秒或几分钟。对于那些不是经常更新，然而一旦有变化就必须立刻更新的资源，例如重大新闻，可以利用各大 CDN 厂商提供的缓存清理（purging）机制处理。这种模式被称为“一直保留，直到被通知”（“Hold’til Told”），意思是永久缓存这些资源，等收到通知后才删除。

6. 条件缓存

如果缓存 TTL 过期，客户端会向服务器发起请求。在多数情况下，收到的响应其实和缓存的版本是一样的，重新下载已经在缓存里的内容也是一种浪费。HTTP 提供条件请求机制，客户端能以有效方式询问服务器："如果内容变了，请返回内容本身；否则，直接告诉我内容没变。"当资源不经常变化时，使用条件请求可以显著节省带宽和性能；但是，保证资源的最新版迅速可用也是非常重要的。使用条件缓存可以通过以下方法。

- 在请求中包含 HTTP 首部 Last-Modified-Since。仅当最新内容在首部中指定的日期之后被更新过，服务器才返回完整内容；否则只返回 304 响应码，并在响应首部中附带上新的时间戳 Date 字段。
- 在请求体中包含实体校验码，或者叫 ETag；它唯一标识所请求的资源。ETag 由服务器提供，内嵌于资源的响应首部中。服务器会比较当前 ETag 与请求首部中收到的 ETag，如果一致，就只返回 304 响应码；否则返回完整内容。

一般来说，大多数 Web 服务器会对图片和 CSS/JS 使用这些技术；但是你应该检查一下，它们是否也对其他可缓存资源起作用。

7. 压缩和代码极简化

所有的文本内容（HTML、JS、CSS、SVG、XML、JSON、字体等），可以从压缩和极简化中受益。这两种方法组合起来，可以显著减少资源大小。更少字节数对应着更少的请求－应答，也就意味着更短的请求时间。

极简化（minification）是指从文本资源中剥离所有非核心内容的过程。通常，这些内容是开发人员敲出来的，所以要考虑方便人类阅读和维护。尽管如此，浏览器并不关心可读性，放弃**代码可读性**反而能节省空间。举个简单例子，看看如下的 HTML：

```
<html>
<head>
  <!-- Change the title as you see fit -->
  <title>My first Web page</title>
</head>
<body>
<!-- Put your message of the day here -->
<p>Hello, World!</p>
</body>
</html>
```

这是一个完全合法的 HTML 页面，可以在浏览器中完美渲染出来（真是废话）。但是，这里面有些信息并不是浏览器所需要的，包括注释、换行、空格。极简化的版本可能看起来像这样：

```
<html><head><title>My first web page</title></head><body>
<p>Hello, World!</p></body></html>
```

没有之前那么美观，也不太容易维护；但是，它的字节数少了一半（之前是 186 字节，现在只要 92 字节）。

在极简化的基础上，压缩可以进一步减少字节数。它通过可无损还原的算法减少资源大小。在发送资源之前，如果服务器进行压缩处理，可以节省 90% 的大小。常规的压缩算法包括 gzip 和 deflate；相对晚些面世的 Brotli 算法也开始崭露头角了。

8. 避免阻塞CSS/JS

CSS 的作用是告诉浏览器以什么方式在可视区域的哪个部分渲染内容。所以，在屏幕上绘制第一个像素之前，浏览器必须确保 CSS 已经下载完整。尽管浏览器的预处理器很智能，会尽早请求整个页面所需要的 CSS，但是把 CSS 资源请求放在页面靠前的部分仍然是种最佳实践，具体位置是在文档的 head 标签里，而且要在任何 JS 或图片被请求和处理之前。

默认情况下，如果在 HTML 中定位了 JS，它就会被请求、解析，然后执行。在浏览器处理完这个 JS 之前，会阻止其后任何资源的下载渲染。有时候，用特定 JS 的下载和执行来阻塞解析其他 HTML 代码是可取的。例如，用它来实例化一个所谓的标签管理器（tag-manager），或者 JS 应被首先执行的其他重要情况，比如用来避免引用不存在的元素或避免竞争条件。

然而大多数时候，这种默认的阻塞行为导致了不必要的延迟，甚至会造成单点故障。为了减轻 JS 阻塞带来的潜在影响，下面针对己方资源（你能控制的）和第三方资源（你不能控制的）推荐了不同的策略。

- 定期校验这些资源的使用情况。随着时间的变迁，Web 页面可能会持续下载一些不再需要的 JS；这时候，最快速有效的解决办法就是去掉它。
- 如果 JS 执行顺序无关紧要，并且必须在 onload 事件触发之前运行，那么可以设置 `async` 属性[7]，像这样：

  ```
  <script async src="/js/myfile.js">
  ```

  只需做到下载 JS 与解析 HTML 并行，就能极大地提升整体用户体验。请慎用 `document.write` 指令，因为很可能中断页面执行，所以需要仔细测试。

- 如果 JS 执行顺序很重要，并且你也能承受脚本在 DOM 加载完之后运行，那么请使用 `defer` 属性。像这样：

  ```
  <script defer src="/js/myjs.js">
  ```

- 对不会影响到页面初次展示的 JS 脚本，必须在 onload 事件触发之后请求（处理）它。

注 7：http://caniuse.com/#search=async

- 如果你不想延迟主页面的 onload 事件，可以考虑通过 iframe 获取 JS，因为它的处理独立于主页面。但是，通过 iframe 下载的 JS 访问不了主页面上的元素。

如果以上这些技巧听起来有点复杂，那是因为本来就是如此。这个问题没有万全之策；而且在不了解业务诉求和 HTML 完整上下文的情况下，盲目推荐某种特定方案是有风险的。尽管如此，如果要阻止 JS 毫无理由地阻塞页面渲染，上述各条仍然是不错的着手点。

9. 图片优化

对大多数网站而言，图片的相对重要性和绝对重要性都在不断增加。图 3-6 展示了过去 5 年间每个 Web 页面的平均请求数和字节大小。[8]

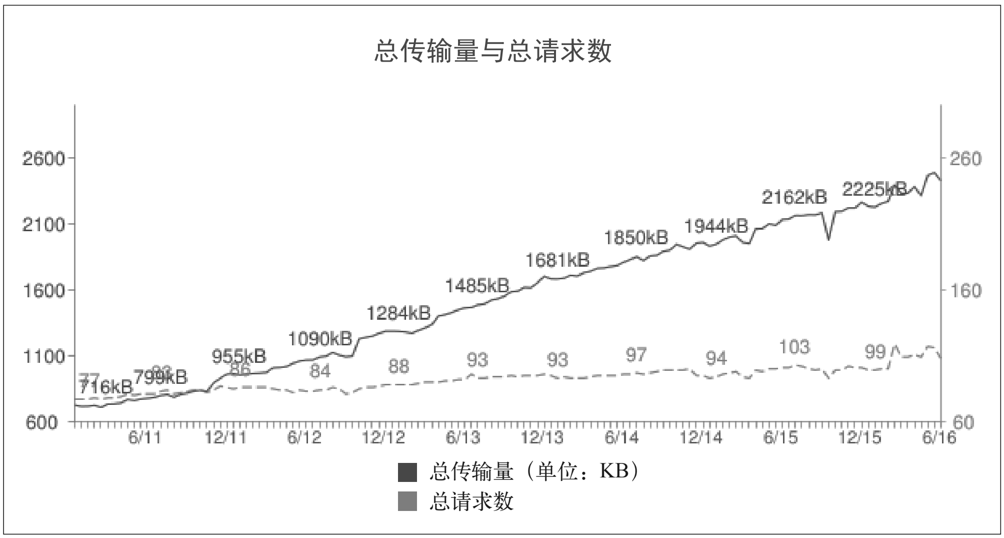


图 3-6：2011—2016 年间页面传输大小和包含请求数（来源：httparchive.com）

既然图片主导了多数现代网站，优化它们就能够获得最大的性能回报。所有图片优化手段的目标都是在达到指定视觉质量的前提下传输最少的字节。这一目标会受很多因素影响，应该引起重视。

- 图片元信息，例如题材地理位置信息、时间戳、尺寸和像素信息，通常包含在二进制数据里，应该在发送给客户端之前去掉（务必保留版权和色彩描述信息）。这种无损处理能够在图片生成时完成。对于 PNG 图片，一般会节省大概 10% 的空间。如果你想学习更多图片优化技巧，可以阅读由 Tim Kadlec、Colin Bendell、Mike McCall、Yoav Weiss、Nick Doyle 和 Guy Podjarny 合著的 *High Performance Images*（O'Reilly 出版社，2016 年）。

注 8：http://httparchive.org/trends.php?s=All&minlabel=Dec+16+2010&maxlabel=Jun+15+2016#bytesTotal&reqTotal

- 图片过载（image overloading）是指，图片最终被浏览器自动缩小，要么因为原始尺寸超过了浏览器可视区中的占位大小，要么因为像素超过设备的显示能力。这不仅浪费带宽，消耗的 CPU 资源也很可观，这些计算资源有时在手持设备上相当宝贵。在响应式设计（RWD）的 Web 站点上，这个问题比较常见，因为无论在什么设备上渲染，都始终返回同样尺寸的图片。图 3-7 反映了这一问题。

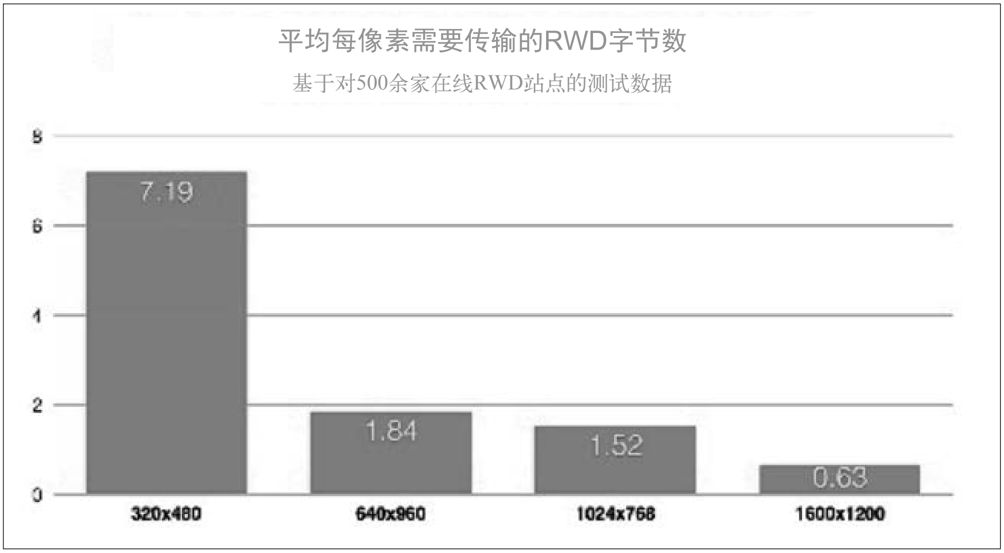


图 3-7：平均每像素需要传输的 RWD 字节数。来源：http://goo.gl/6hOkQp

想要解决图片过载，可以使用技术手段，针对用户的设备、网络状况和预期的视觉质量，提供裁剪过的图片（就尺寸和质量而言）。

3.2.2 反模式

HTTP/2 对每个域名只会开启一个连接，所以 HTTP/1.1 下的一些诀窍对它来说只会适得其反。接下来讨论几个如今流行却不再适用于 h2 站点的做法。

1. 生成精灵图和资源合并/内联

精灵图（spriting）是指把很多小图片拼合成一张大图，这样只需发起一个请求就可以覆盖多个图片元素。例如，颜色样本或导航元素（箭头、图标等）拼合成一张大的图片，这就是精灵图。在 HTTP/2 中，针对特定资源的请求不再是阻塞式的，很多请求可以并行处理；于是就性能而言，生成精灵图就失去意义了，网站管理员不再需要考虑创建它们，虽然已经创建的精灵图不一定非要取消。

与之类似，小的文本资源，例如 JS 和 CSS，会依照惯例合并成一份更大的资源，或者直接内嵌在主体 HTML 中，这也是为了减少客户端－服务器连接数。这种做法有个问题是，

那些小的 CSS 或 JS 自身也许可缓存，但如果它们内嵌在不可缓存的 HTML 中的话，当然也就不可缓存了。所以，把站点从 h1 迁移到 h2 时，要避免这些做法。不过，2015 年 11 月，khanacademy.org 发表的一份研究报告[9] 指出，把很多小的 JS 脚本合并成一个大文件可能仍旧对 h2 有意义，因为这样可以更好地压缩处理并节省 CPU。

2. 域名拆分

域名拆分（sharding）是为了利用浏览器针对每个域名开启多个连接的能力来并行下载资源。对于某个具体的站点，找到最适合拆分的域名个数并没有什么公式；可以这么说，各种观点都在业内流行。

在 HTTP/2 的领域，网站管理员反而需要花费可观的精力在收拢域名上。比较好的办法就是继续保持当前的域名拆分，但是确保这些域名共享同一张证书 [通配符 / 存储区域网络（SAN）]，并保持服务器 IP 地址和端口相同，以便从浏览器网络归并（network coalescence）中收益，这样可以节省为单个域名连接建立的时间。

3. 禁用cookie的域名

在 HTTP/1 下，请求和响应首部从不会被压缩。随着时间推移，首部大小已经增长了，超过单个 TCP 数据包（约 1.5KB）的 cookie 可以说司空见惯。因此，在内容源和客户端之间来回传输首部信息的开销可能造成明显的延迟。

因此，对图片之类不依赖于 cookie 的资源，设置禁用 cookie 的域名是个合理的建议。

但是 HTTP/2 中，首部是被压缩的（参见 5.6 节），并且客户端和服务器都会保留“首部历史”，避免重复传输已知信息。所以，如果你要重构站点，大可不必考虑禁用 cookie 的域名，这样能减少很多包袱。

静态资源也应该从同一域名提供；使用与主页面 HTTP 相同的域名，消除了额外的 DNS 查询以及（潜在的）socket 连接，它们都会减慢静态资源的获取。把阻塞渲染的资源放在同样的域名下，也可以提升性能。

3.3　小结

HTTP/1.1 孕育了一个混乱不堪或者称得上是冒险刺激的世界，包含了各种性能优化手段与诀窍。业界人士挖空心思追求性能，由此带来的混乱已经登峰造极。HTTP/2 的目标之一就是淘汰掉众多（并不是全部）此类诀窍。不管怎样，了解这些手段及其运行机制，可以帮助我们深入理解 Web 及其内部实现。

注 9：http://engineering.khanacademy.org/posts/js-packaging-http2.htm

第 4 章

HTTP/2迁移

一个人可以不负责地说，要支持 HTTP/2，你要做的无非是升级到支持 h2 的 Web 服务器，或者在你的网站前面加上支持 h2 的 CDN。这么说没错，但它可能掩盖了大量微妙的细节，导致意料之外的较高成本和次优性能。在升级到 HTTP/2 之前，你应该考虑如下方面：

- 浏览器对 h2 的支持情况
- 迁移到 TLS（HTTPS）的可能性
- 对你的网站做基于 h2 的优化（可能对 h1 有反作用）
- 网站上的第三方资源
- 保持对低版本客户端的兼容

本章将会对上述各个方面进行详细讲解。

4.1　浏览器的支持情况

在本书出版的时候，大约 80% 的浏览器可以在一定程度上支持 h2。[1] 这意味着，只要网站开启 h2，相当一部分人就可以开始享受 h2 带来的益处。由于协议协商对不支持 h2 的浏览器是透明的，任何不支持 h2 的客户端都将简单地退回到 h1，并仍然可以访问你的站点基础设施。（现在你还保留着对 h1 的支持，对吧？）

表 4-1 列举了一些浏览器，及其最早开始支持 HTTP/2 的版本。

注 1：http://caniuse.com/#search=http2

表4-1：HTTP/2浏览器支持情况

浏览器名称	最低支持版本	备注
Chrome	41	
Firefox	36	
Edge	12	
Safari	9	OSX 10.11 及之后的版本
Internet Explorer	11	仅支持 Windows 10
Opera	28	
Safari - iOS	9.2	
Android Browser	51	
Chrome - Android	51	

4.2 迁移到TLS

所有主流浏览器只能访问基于 TLS（即 HTTPS 请求）的 h2，这就意味着你需要支持 TLS，否则游戏就没法玩了。不仅如此，TLS 本身的要求也很高：你需要支持 TLS 1.2 或更高版本，以及一组特定的临时加密算法（更多信息参见 9.2 节中关于 RFC 7540 的说明）。因为大多数关注安全的现代网站已经被“TLS 无处不在”的大潮所席卷，所以这对你应该不成问题；但是如果成问题的话，就别吝啬时间和资源了。

> 本节并不打算充当使用 TLS 加密网站的入门指导。这个主题已经有很多图书详细介绍过了。管理网站安全是个细致活儿，一不注意就会遇到很多陷阱。请花时间研究你的网站所依赖的工具。关于 TLS 和证书的入门经典，可以参考 Ilya Grigorik 的著作《Web 性能权威指南》[2]。

在过去 5 年左右的时间里，人们投入了大量的精力，尽可能降低网站迁移和启用 TLS 的痛苦，但仍然有些额外的工作是免不了的。需要考虑的事情如下。

了解你使用的Web服务器

每种 Web 服务器配置 HTTPS 的方式都略有差异。

获得证书

EV、OV、DV、CN、CSR、X509、SAN——你得分清这些缩略词。获取网站的证书通常要完成几个步骤，包括创建证书签名请求（CSR），验证你的身份和证书所有者，然后从证书颁发机构（CA）购买证书。有很多 CA 可供选择。Let’s Encrypt[3] 等机构致力于让获取证书变得简单、快捷，甚至免费。

注 2：该书已由人民邮电出版社出版，书号：9787115349101。——编者注

注 3：https://letsencrypt.org/

保护私钥

证书是否安全完全取决于你自己。为了使网站安全地基于 TLS 运行，你应该考虑私钥存放的方式、位置，以及哪些人有访问权限。解决方案有很多种，可以使用价格高昂的硬件安全模块（HSM），也可以除了祈祷什么都不做，还有一些其他诀窍。如果你不熟悉 TLS，就得在工作计划中重视它。

为增加的服务器负载做准备

大家已经做了很多努力来降低 TLS 的性能消耗，但这个游戏的正常节奏就是进三步退两步。虽然对称加密算法的优化已经帮了大忙，但是采用临时密钥交换又抵消了这种好处（虽然它让一切更安全）。下面这些诀窍可供借鉴。

- 保持连接开启尽可能长的时间。TLS 成本最高的地方就是连接时的握手过程。如果连接时间能保持足够长，握手次数就可以减少。
- 使用会话凭证。会话凭证允许客户端复用上次握手时完成的复杂的加解密计算结果，直接重连服务器。
- 使用内置加解密支持的芯片。Intel 现代处理器上拥有的 AES-NI[4] 指令，能大大加快对称加解密的速度。

紧跟潮流

Web 安全是不断变化的世界。似乎每隔几个月，服务器和 HTTPS 就会曝出新的漏洞。所以，紧跟最新、最伟大的变化非常重要，这会避免已有成果在未来迅速沦为摆设。

定期检测

应该使用工具定期检测网站的 TLS 配置，例如 Qualys Lab 的 SSL Test[5]。

只要服务基于 TLS，这些诀窍就值得铭记于心。但是，TLS 的功能异常丰富，得投入大量的时间才能彻底弄清楚。所以请记住：

> 一知半解是危险的；如果不打算开怀畅饮，就别去品尝知识的泉水。
>
> ——亚历山大·蒲柏

注 4：https://www.intel.com/content/www/us/en/architecture-and-technology/advanced-encryption-standard--aes-/data-protection-aes-general-technology.html

注 5：https://www.ssllabs.com/

> **TLS 是必要的吗**
>
> 简而言之，并不是。不过有用的答案是肯定的。虽然 HTTP/2 的规范并不明确**要求** TLS，也支持以明文通信，但主流浏览器都不支持基于非 TLS 的 h2。这背后有两个原因。
>
> 首先，一个非常现实的原因是，从之前对 WebSocket 和 SPDY 的实验看来，使用 Upgrade 首部，通过 80 端口（**明文**的 HTTP 端口）通信时，通信链路上代理服务器的中断等因素会导致非常高的错误率。如果基于 443 端口（HTTPS 端口）上的 TLS 发起请求，错误率会显著降低，并且协议通信也更简洁。第二个原因是，人们越来越相信，考虑到安全和隐私，一切都应该被加密。HTTP/2 被视为一次推动全网加密通信发展的机会。

4.3 撤销针对HTTP/1.1的“优化”

Web 开发者之前花费了大量心血来充分使用 h1，并且已经总结了一些诀窍，例如资源合并、域名拆分、极简化、禁用 cookie 的域名、生成精灵图，等等。所以，当得知这些实践中有些在 h2 下变成反模式时，你可能会感到吃惊。例如，资源合并（把很多 CSS 或 JavaScript 文件拼合成一个）能避免浏览器发出多个请求。对 h1 而言这很重要，因为发起请求的代价很高；但是在 h2 的世界里，这部分已经做了深度优化。放弃资源合并的结果可能是，针对单个资源发起请求的代价很低，但浏览器端可以进行更细粒度的缓存。

表 4-2 列出了一些用于优化 h1 请求的常用技巧，并标注了 h2 方面的考虑。

表4-2：HTTP/1优化技巧，以及HTTP/2的相关建议

名称	描述	备注
资源合并	把多个文件（JavaScript、CSS）合成一个文件，以减少 HTTP 请求	在 HTTP/2 下这并非必要，因为请求的传输字节数和时间成本更低，虽然这种成本仍然存在
极简化	去除 HTML、JavaScript、CSS 这类文件中无用的代码	很棒的做法，在 HTTP/2 下也要保留
域名拆分	把资源分布到不同的域名上面去，让浏览器利用更多的 socket 连接	HTTP/2 的设计意图是充分利用单个 socket 连接，而拆分域名会违背这种意图。建议取消域名拆分，但请注意本表格之后的附注框会介绍这个问题相关的各种复杂情况
禁用 cookie 的域名	为图片之类的资源建立单独的域名，这些域名不用 cookie，以尽可能减少请求尺寸	应该避免为这些资源单独设立域名（参见“域名拆分”），但更重要的是，由于 HTTP/2 提供了首部压缩，cookie 的开销会显著降低
生成精灵图	把多张图片拼合为一个文件，使用 CSS 控制在 Web 页面上展示的部分	与极简化类似，只不过用 CSS 实现这种效果的代价高昂；不推荐在 HTTP/2 中使用

要不要进行域名拆分

HTTP/2 的设计思路是尽量在单个 TCP/IP socket 上通信。它的做法是，开启一个 socket，并以最理想的拥塞速率运行，这样比起协调多个 socket 更可靠也更高效。尽管如此，Akamai 的 Foundry 团队的研究表明，这种策略并不总是有效。[6]取决于网站的具体情况，多个 socket 可能优于单一 socket。它直接依赖于 TCP 拥塞控制的运作方式，以及达到最优设置所需的时间。设置较大的初始拥塞窗口值可以缓解此问题；但是如果这些较大的值无法由通信链路支持，那么也会产生问题。这个例子告诉我们，要充分使用和优化 h2，需要不断地学习。对你的网站而言，就是要开发、测试、调整，以及通过不断重复来找到最优设置。

假设你正花时间为站点采用 h2 做全面优化，有个问题会立刻浮现：仍然有 25% 的访问量来自于 h1 客户端，你也得优化它们访问时的性能。造福所有人是项艰巨的任务。仔细分析自己的用户群，这可能会告诉你应该为哪些人群进行（站点）优化。或者，你可以根据具体情况为 h1 和 h2 用户提供不同的内容，也可以使用 CDN 或类似工具替你优化。[7]

4.4 第三方资源

对第三方资源，我们又爱又恨；然而现实是我们的网站上的确存在第三方内容。第三方内容的问题是，你不能直接控制它。因此，对于这些第三方资源支持什么和不支持什么，你束手无策。如果一切都在单个 socket 上传输，这时候 h2 工作得最好，那还考虑第三方资源干什么？但从现实考虑，第三方资源的确存在，而且会拖累 HTTP/2 带来的任何可能的性能优化。如果你使用的第三方资源不支持 HTTPS，那就更麻烦了。相关研究[8]表明，在很多情况下，第三方资源对页面性能的影响举足轻重。

所以，应该拿第三方内容怎么办？要找到答案，可以从下列问题开始。

- 用到的第三方资源支持 HTTPS 吗？
- 它们是否计划支持 HTTP/2？
- 它们是否意识到，自己应当尽可能降低所提供的资源对页面性能的影响，并将其视为关键任务？

如果上述问题的答案都是否定的，就得考虑两个追加的问题：还有其他第三方资源提供我所需要的东西吗？我究竟是确实需要这些第三方内容，还是没有也行？

注 6：https://www.akamai.com/us/en/multimedia/documents/technical-publication/http2-performance-in-cellular-networks.pdf

注 7：https://blogs.akamai.com/2016/01/how-to-start-optimizing-in-an-h2-world.html

注 8：https://www.akamai.com/us/en/multimedia/documents/technical-publication/are-3rd-parties-slowing-down-the-mobile-web.pdf

4.5 支持旧版本客户端

有些人不喜欢变化，他们现在使用的浏览器对自己来说已经够用了，而且升级浏览器是个麻烦事。问题是，这些人群可能是你的用户 / 客户，你可能也不打算置之不顾。这里有一个真实案例。微软于 2014 年 4 月 8 日终止了对 Windows XP 的支持。这意味着 XP 用户在现代浏览器使用和安全问题上被甩得越来越远。不用说，XP 上的 IE 浏览器不支持 h2；但更重要的是，取决于你的 TLS 设置方式以及是否提供支持 HTTP/1 的备用网站，这些用户可能彻底失去了通过 h1 访问你的内容的机会。一方面，迁移到 HTTP/2 是大势所趋，但是另一方面，你将失去的可能是你的重要用户 / 客户。在迁移到 HTTP/2 之前，现实情况仍然是我们需要考虑的问题。

4.6 小结

虽然迁移到 HTTP/2 通常被视作一件好事，并且你的网站业务理论上完全不受影响，但在迁移之前当然要仔细考虑一些问题。虽然许多主流网站运行 h2 已经有段时间了，但这并不意味着采用 HTTP/2 会“稳赢”。对待它应当和对待其他重大变化一样：测试，测试，再测试。

第 5 章

HTTP/2 协议

本章全面探讨了 HTTP/2 的底层工作原理，深入到数据层传输的帧及其通信方式。这将帮你理解协议提供的许多益处和问题。读完本章后，你应该能掌握足够的信息来调试和优化自己的 h2 安装设置，以便充分利用协议。如果你是勇士，希望进一步深入协议或靠自己来实现，RFC 7540[1] 会是很好的起点。

5.1 HTTP/2分层

HTTP/2 大致可以分为两部分：**分帧层**，即 h2 多路复用能力的核心部分；**数据**或 http 层，其中包含传统上被认为是 HTTP 及其关联数据的部分。彻底分开这两层，把它们当成彼此独立的事物，这是非常诱人的想法。仔细阅读 HTTP/2 规范的读者会发现，分帧层究竟是被设计成完全通用的、可重用的数据结构，还是用来传输 HTTP 内容，二者是有些分歧的。比如，规范起初泛泛地讨论了终端和双向通信——这对许多消息系统来讲是完美无缺的——然后话锋一转，讨论起了客户端、服务器、请求和响应。读到分帧层的时候，不要忘记这一事实：它的目的是传输 HTTP，而不是其他。

尽管数据层被设计成可以向后兼容 HTTP/1.1，对于熟悉 h1 并习惯于阅读线上协议的开发者来说，还有些地方需要重新确认。

二进制协议

h2 的分帧层是基于帧的二进制协议。这方便了机器解析，但是肉眼识别起来比较困难。

注 1：https://tools.ietf.org/html/rfc7540

首部压缩

仅仅使用二进制协议似乎还不够，h2 的首部还会被深度压缩。这将显著减少传输中的冗余字节。

多路复用

在你喜爱的调试工具里查看基于 h2 传输的连接的时候，你会发现请求和响应交织在一起。

加密传输

最重要的是，线上传输的绝大部分数据是加密过的，所以在中途读取会更加困难。

现在，我们来展开这些话题。

5.2 连接

连接是所有 HTTP/2 会话的基础元素，其定义是客户端初始化的一个 TCP/IP socket，客户端是指发送 HTTP 请求的实体。这和 h1 是一样的，不过与完全无状态的 h1 不同的是，h2 把它所承载的帧（frame）和流（stream）共同依赖的连接层元素捆绑在一起，其中既包含连接层设置也包含首部表（稍后有对两者更详细的描述）。也就是说，与之前的 HTTP 版本不同，每个 h2 连接都有一定的开销。之所以这么设计，是考虑到收益远远超过其开销。

是否支持 h2

协议发现——识别终端是否支持你想使用的协议——会比较棘手。HTTP/2 提供两种协议发现的机制。

在连接不加密的情况下，客户端会利用 Upgrade 首部来表明期望使用 h2。如果服务器也可以支持 h2，它会返回一个“101 Switching Protocols”（协议转换）响应。这增加了一轮完整的请求－响应通信。

如果连接基于 TLS，情况就不同了。客户端在 `ClientHello` 消息中设置 ALPN（Application-Layer Protocol Negotiation，应用层协议协商）扩展来表明期望使用 h2 协议，服务器用同样的方式回复。如果使用这种方式，那么 h2 在创建 TLS 握手的**过程中**完成协商，不需要多余的网络通信。值得注意的是，SPDY 和 h2 的早期修订版本使用 NPN（Next Protocol Negotiation，下一代协议协商）扩展来完成 h2 协商。它在 2014 年中期被 ALPN 取代。

表明终端支持 h2 的最后一个方法是使用 HTTP Alternative Services（HTTP 替代服务）[2] 或 Alt-Svc。 服务器可以用这种办法，在返回给客户端的响应首部中，表示后续的请求或许可以使用更合适的请求地址或协议。 这个工具非常灵活，浏览器的支持也在不断增加。 它不是用来替代 ALPN 的，但它是值得特别关注。

注 2：https://tools.ietf.org/html/rfc7838

为了向服务器双重确认客户端支持 h2，客户端会发送一个叫作 connection preface（连接前奏）的魔法字节流，作为连接的第一份数据。这主要是为了应对客户端通过纯文本的 HTTP/1.1 升级上来的情况。该字节流用十六进制表示如下：

```
0x505249202a20485454502f322e300d0a0d0a534d0d0a0d0a
```

解码为 ASCII 是：

```
PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n
```

这个字符串的用处是，如果服务器（或者中间网络设备）不支持 h2，就会产生一个显式错误。这个消息特意设计成 h1 消息的样式。如果运行良好的 h1 服务器收到这个字符串，它会阻塞这个方法（PRI）或者版本（HTTP/2.0），并返回错误，可以让 h2 客户端明确地知道发生了什么错误。

这个魔法字符串会有一个 SETTINGS 帧紧随其后。服务器为了确认它可以支持 h2，会声明收到客户端的 SETTINGS 帧，并返回一个它自己的 SETTINGS 帧（反过来也需要确认），然后确认环境正常，可以开始使用 h2。大家做了很多工作，保证这个流程尽可能高效。虽然表面上看起来有点啰嗦，但客户端可以立即开始发送帧，并假设服务器的 SETTINGS 帧已经到了。 如果在偶然情况下，过份乐观的客户端在 SETTINGS 帧之前收到一些数据，那么协商会失败，客户端和服务端都会收到 GOAWAY 帧。

隐藏的信息

连接前奏包含两条“秘密”信息。 第一条是关于美国国家安全局 PRISM（棱镜）监控计划的一个笑话。 HTTP/2 还处于发展早期时，恰好这份计划公开曝光，于是有些聪明人决定借协议让大家铭记这份计划。(你还认为我们协议开发人员没有幽默感吗？）第二条涉及 HTTP/2.0 的名称。 在协议制定过程中，很早就把小数点去掉了，这表明未来的 HTTP 版本不能保证语义的向后兼容[3]。 然而，更早的版本一直没有去掉它。 因此，本书中出现 HTTP/2.0 的地方，主要是考虑到历史准确性和上下文的需要。

5.3 帧

之前说过，HTTP/2 是基于帧（frame）的协议。采用分帧是为了将重要信息都封装起来，让协议的解析方可以轻松阅读、解析并还原信息。 相比之下，h1 不是基于帧的，而是以文本分隔。 看看下面的简单例子：

```
GET / HTTP/1.1 <crlf>
```

注 3：因为这意味着不会有 2.1、2.2 之类的版本。——译者注

```
Host: www.example.com <crlf>
Connection: keep-alive <crlf>
Accept: text/html,application/xhtml+xml,application/xml;q=0.9... <crlf>
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_4)... <crlf>
Accept-Encoding: gzip, deflate, sdch <crlf>
Accept-Language: en-US,en;q=0.8 <crlf>
Cookie: pfy_cbc_lb=p-browse-w; customerZipCode=99912|N; ltc=%20;...<crlf>
<crlf>
```

解析这种数据用不着什么高科技，但往往速度慢且容易出错。你需要不断读入字节，直到遇到分隔符为止（这里是指 <crlf>），同时还要考虑一些不太守规矩的客户端，它们会只发送 <lf>。于是大概需要这样一台状态机：

```
loop
 while( ! CRLF )
  read bytes
 end while

 if first line
   parse line as the Request-Line
 else if line is empty
   break out of the loop # 完成
 else if line starts with non-whitespace
   parse the header line into a key/value pair
 else if line starts with space
   add the continuation header to the previous header
 end if
end loop

# 好了,准备根据Transfer-encoding首部的值处理请求和响应,还有各种浏览器bug吧
```

这样写程序是可行的，并且这事已经做过无数次了。解析 h1 的请求或响应可能出现下列问题。

- 一次只能处理一个请求或响应，完成之前不能停止解析。
- 无法预判解析需要多少内存。这会带来一系列问题：你要把一行读到多大的缓冲区里；如果行太长会发生什么；应该增加并重新分配内存，还是返回 400 错误。为了解决这些问题，保持内存处理的效率和速度可不简单。

从另一方面来说，有了帧，处理协议的程序就能预先知道会收到什么。基于帧的协议，特别是 h2，开始有固定长度的字节，其中包含表示整帧长度的字段。图 5-1 是一个 HTTP/2 帧的结构。

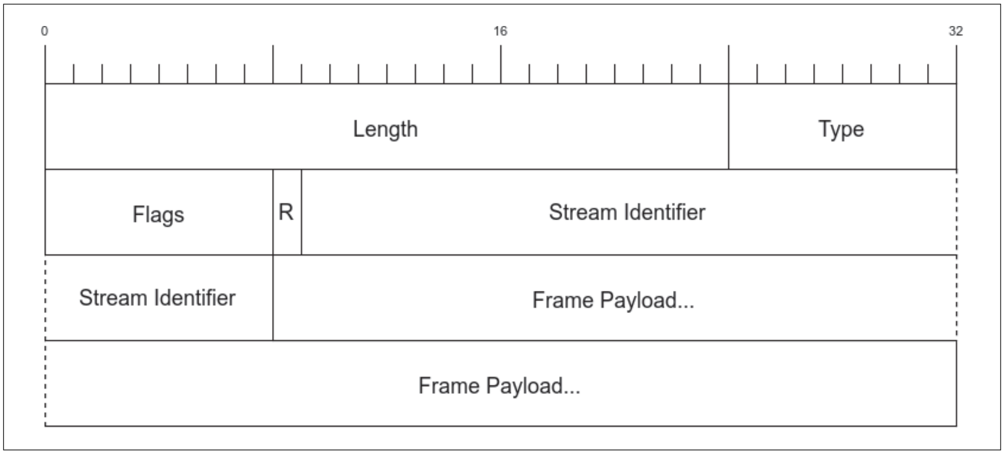


图 5-1：HTTP/2 帧结构

前 9 个字节对于每个帧是一致的。解析时只需要读取这些字节，就可以准确地知道在整个帧中期望的字节数。其中每个字段的说明，参见表 5-1。

表5-1：帧首部字段

名称	长度	描述
Length	3 字节	表示帧负载的长度（取值范围为 2^{14}~2^{24}−1 字节）。 请注意，2^{14} 字节是默认的最大帧大小，如果需要更大的帧，必须在 SETTINGS 帧中设置
Type	1 字节	当前帧类型（见表 5-2 中介绍）
Flags	1 字节	具体帧类型的标识
R	1 位	保留位，不要设置，否则可能带来严重后果
Stream Identifier	31 位	每个流的唯一 ID
Frame Payload	长度可变	真实的帧内容，长度是在 Length 字段中设置的

因为规范严格明确，所以解析逻辑大概是这样：

```
loop
 Read 9 bytes off the wire                           // 读前9字节
 Length = the first three bytes                      // 长度值为前3字节
 Read the payload based on the length.               // 基于长度读负载
 Take the appropriate action based on the frame type. // 根据帧类型采取对应操作
end loop
```

这样一来，实现和维护都会简单很多。相比依靠分隔符的 h1，h2 还有另一大优势：如果使用 h1 的话，你需要发送完上一个请求或者响应，才能发送下一个；由于 h2 是分帧的，请求和响应可以交错甚至多路复用。多路复用有助于解决类似队头阻塞的问题，具体描述见第 3 章。

h2 协议中有 10 种不同的帧类型。概览见表 5-2，具体细节在附录 A 中讲解。

表5-2：HTTP/2帧类型

名称	ID	描述
DATA	0x0	传输流的核心内容
HEADERS	0x1	包含 HTTP 首部，和可选的优先级参数
PRIORITY	0x2	指示或者更改流的优先级和依赖
RST_STREAM	0x3	允许一端停止流（通常由于错误导致的）
SETTINGS	0x4	协商连接级参数
PUSH_PROMISE	0x5	提示客户端，服务器要推送些东西
PING	0x6	测试连接可用性和往返时延（RTT）
GOAWAY	0x7	告诉另一端，当前端已结束
WINDOW_UPDATE	0x8	协商一端将要接收多少字节（用于流量控制）
CONTINUATION	0x9	用以扩展 HEADER 数据块

可扩展空间

HTTP/2 内置了名为**扩展帧**的处理新的帧类型的能力。依靠这种机制，客户端和服务器的实现者可以实验新的帧类型，而无需制定新协议。按照规范，任何客户端不能识别的帧都会被丢弃，所以网络上新出现的帧就不会影响核心协议。当然，如果你的应用程序依赖于新的帧，而中间代理会丢弃它，那么可能会出现问题。

5.4 流

HTTP/2 规范对流（stream）的定义是："HTTP/2 连接上独立的、双向的帧序列交换。"你可以将流看作在连接上的一系列帧，它们构成了单独的 HTTP 请求和响应。如果客户端想要发出请求，它会开启一个新的流。然后，服务器将在这个流上回复。这与 h1 的请求 / 响应流程类似，重要的区别在于，因为有分帧，所以多个请求和响应可以交错，而不会互相阻塞。流 ID（帧首部的第 6~9 字节）用来标识帧所属的流。

客户端到服务器的 h2 连接建立之后，通过发送 HEADERS 帧来启动新的流，如果首部需要跨多个帧，可能还发会送 CONTINUATION 帧（更多信息参见下面的附注栏"CONTINUATIONS 帧"）。该 HEADERS 帧可能来自 HTTP 请求，也可能来自响应，具体取决于发送方。后续流启动的时候，会发送一个带有递增流 ID 的新 HEADERS 帧。

> **CONTINUATION 帧**
>
> HEADERS 帧通过在帧的 `Flags` 字段中设置 END_HEADERS 标识位来标识首部的结束。在单个 HEADERS 帧装不下所有 HTTP 首部的情况下（例如，帧可能比当前最大长度还长），不会设置 END_HEADERS 标识位，而是在之后跟随一个或多个 CONTINUATION 帧。我们可以把 CONTINUATION 帧当作特殊的 HEADERS 帧。那么，为什么要使用特殊的帧，而不是再次使用 HEADERS 帧？ 如果重复使用 HEADERS，那么后续的 HEADERS 帧的负载就得经过特殊处理才能和之前的拼接起来。这些帧首部是否需要重复？这样的话，如果帧之间存在分歧该怎么办？协议开发者不喜欢这类模棱两可的情况，因为它可能在未来引起麻烦。考虑到这一点，工作组决定增加一个明确的帧类型，以避免实现混淆。
>
> 需要注意的是，由于 HEADERS 和 CONTINUATION 帧必须是有序的，使用 CONTINUATION 帧会破坏或减损多路复用的益处。CONTINUATION 帧是解决重要场景（大首部）的工具，但只能在必要时使用。

5.4.1 消息

HTTP 消息泛指 HTTP 请求或响应。上一节已经讲过，流是用来传输一对请求 / 响应消息的。一个消息至少由 HEADERS 帧（它初始化流）组成，并且可以另外包含 CONTINUATION 和 DATA 帧，以及其他的 HEADERS 帧。 图 5-2 是普通 GET 请求的示例流程。

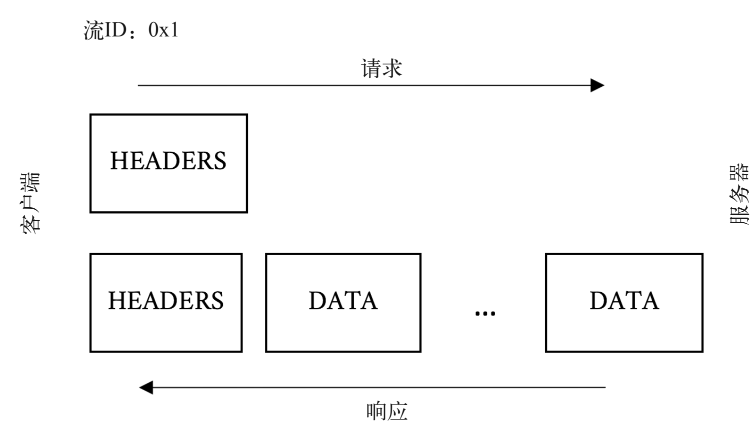


图 5-2：GET 请求和响应消息

图 5-3 展示了某个 POST 消息对应的各帧可能的样子。请注意，POST 和 GET 的主要差别之一就是 POST 请求通常包含客户端发出的大量数据。

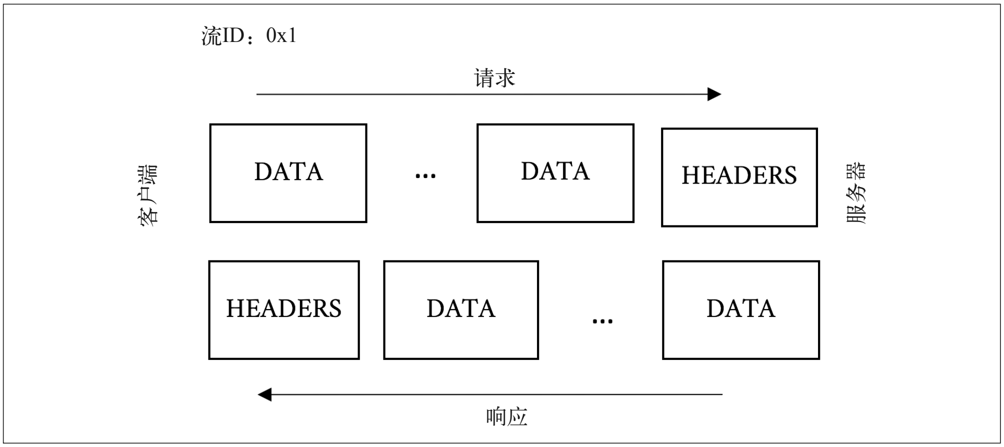


图 5-3：Post 请求的请求和响应消息

h1 的请求和响应都分成消息首部和消息体两部分；与之类似，h2 的请求和响应分成 HEADERS 帧和 DATA 帧。

HTTP 消息是在 HTTP/1.1 的 RFC 7230[4] 中定义的，此处供参考。

HTTP/1 和 HTTP/2 消息的下列差别是需要注意的。

一切都是header

h1 把消息分成两部分：请求 / 状态行；首部。h2 取消了这种区分，并把这些行变成了魔法**伪**首部。举个例子，HTTP/1.1 的请求和响应可能是这样的：

```
GET / HTTP/1.1
Host: www.example.com
User-agent: Next-Great-h2-browser-1.0.0
Accept-Encoding: compress, gzip

HTTP/1.1 200 OK
Content-type: text/plain
Content-length: 2
...
```

在 HTTP/2 中，它等价于：

```
:scheme: https
:method: GET
:path: /
:authority: www.example.com
User-agent: Next-Great-h2-browser-1.0.0
Accept-Encoding: compress, gzip
```

注 4：https://tools.ietf.org/html/rfc7230

```
:status: 200
content-type: text/plain
```

请注意，请求和状态行在这里拆分成了多个首部，即 `:scheme`、`:method`、`:path` 和 `:status`。同时要注意的是，h2 的这种表示方式跟数据传输时不同。想了解更多信息的话，可以翻到附录 A 的 A.3 节查看 HEADERS 帧的描述，5.6 节也有相关内容可供参考。

没有分块编码（chunked encoding）

在基于帧的世界里，谁还需要分块？只有在无法预先知道数据长度的情况下向对方发送数据时，才会用到分块。在使用帧作为核心协议的 h2 里，就不再需要它了。

不再有101的响应

Switching Protocol 响应是 h1 的边缘应用。它如今最常见的应用可能就是用以升级到 WebSocket 连接。ALPN 提供了更明确的协议协商路径，往返的开销也更小。

5.4.2 流量控制

h2 的新特性之一是基于流的流量控制。不同于 h1 的世界，只要客户端可以处理，服务端就会尽可能快地发送数据，h2 提供了客户端调整传输速度的能力。（并且，由于在 h2 中，一切几乎都是对称的，服务端也可以调整传输的速度。）WINDOW_UPDATE 帧用来指示流量控制信息。每个帧告诉对方，发送方想要接收多少字节。当一端接收并消费被发送的数据时，它将发出一个 WINDOW_UPDATE 帧以指示其更新后的处理字节的能力。（许多早期的 HTTP/2 实现者花了大量时间调试窗口更新机制，来回答“为什么我没有取到数据”的问题。）发送方有责任遵守这些限制。

客户端有很多理由使用流量控制。一个很现实的原因可能是，确保某个流不会阻塞其他流。也可能客户端可用的带宽和内存比较有限，强制数据以可处理的分块来加载反而可以提升效率。尽管流量控制不能关闭，把窗口最大值设定为设置 $2^{31}-1$ 就等效于禁用它，至少对小于 2GB 的文件来说是如此。另一个需要注意的是中间代理。通常情况下，网络内容通过代理或者 CDN 来传输，也许它们就是传输的起点或终点。由于代理两端的吞吐能力可能不同，有了流量控制，代理的两端就可以密切同步，把代理的压力降到最低。

流量控制示例

在流建立的时候，窗口大小默认都是 65 535（$2^{16}-1$）字节。假设客户端 A 支持该默认值，它的另一端（B）发送了 10 000 字节，B 也会关注窗口大小（现在有 55 535 字节了）。现在 A 花时间处理了 5000 字节，还剩下 5000 字节，然后它会发送一个 WINDOW_UPDATE 帧，说明它现在的窗口大小是 60 535 字节。B 收到这个帧之后，开始发送一个大文件（比如 4GB 大小）。在这个场景下，在 B 等 A 准备好接收更多的数据之前，B 能发送的数据量就是当前窗口的大小，即 60 535 字节。通过这种方式，A 可以控制 B 发送数据的最大速率。

5.4.3 优先级

流的最后一个重要特性是依赖关系。现代浏览器都经过了精心设计，首先请求网页上最重要的元素，以最优的顺序获取资源，由此来优化页面性能。拿到了 HTML 之后，在渲染页面之前，浏览器通常还需要 CSS 和关键 JavaScript 这样的东西。在没有多路复用的时候，在它可以发出对新对象的请求之前，需要等待前一个响应完成。有了 h2，客户端就可以一次发出所有资源的请求，服务端也可以立即着手处理这些请求。由此带来的问题是，浏览器失去了在 h1 时代默认的资源请求优先级策略。假设服务器同时接收到了 100 个请求，也没有标识哪个更重要，那么它将几乎同时发送每个资源，次要元素就会影响到关键元素的传输。

h2 通过流的依赖关系来解决这个问题。通过 HEADERS 帧和 PRIORITY 帧，客户端可以明确地和服务端沟通它需要什么，以及它需要这些资源的顺序。这是通过声明**依赖关系**树和树里的相对**权重**实现的。

- **依赖关系**为客户端提供了一种能力，通过指明某些对象对另一些对象有依赖，告知服务器这些对象应该优先传输。
- **权重**让客户端告诉服务器如何确定具有共同依赖关系的对象的优先级。

来看下面这个简单的网站：

- index.html
  - header.jpg
  - critical.js
  - less_critical.js
  - style.css
  - ad.js
  - photo.jpg

在收到主体 HTML 文件之后，客户端会解析它，并生成依赖树，然后给树里的元素分配权重。这时这棵树可能是这样的：

- index.html
  - style.css
    - critical.js
      - less_critical.js (weight 20)
      - photo.jpg (weight 8)
      - header.jpg (weight 8)
      - ad.js (weight 4)

在这个依赖树里，客户端表明它最需要的是 style.css，其次是 critical.js。没有这两个文件，它就不能接着渲染页面。等它收到了 critical.js，就可以给出其余对象的相对权重。权重表示服务一个对象时所需要花费的对应“努力”程度。这个例子中，less_critical.js 的权重为 20，而所有元素的权重之和为 40。也就是说，服务器应当花费大约一半的时间或资源用以传输 less_critical.js，其他三个占了另外一半。称职的服务器会尽最大努力确保客户端尽快获得这些对象。不过说到底，做什么以及如何处理优先级，还是得听服务器的。它仍有做它自己认为正确的事的权力。 处理优先级的智能水平，可能会是决定各种支持 h2 的 Web 服务器性能优劣的重要因素。

5.5 服务端推送

提升单个对象性能的最佳方式，就是在它被用到之前就放到浏览器的缓存里面。这正是 HTTP/2 的服务端推送的目的。推送使服务器能够主动将对象发给客户端，这可能是因为它知道客户端不久将用到该对象。如果允许服务器随意地将对象发送给客户端，可能会产生包括性能和安全在内的一系列问题，因此它不仅仅是一个如何做的问题，还是一个如何做才对的问题。

5.5.1 推送对象

如果服务器决定要推送一个对象（RFC 中称为“推送响应”），会构造一个 PUSH_PROMISE 帧。这个帧有很多重要属性，列举如下。

- PUSH_PROMISE 帧首部中的流 ID 用来响应相关联的请求。推送的响应一定会对应到客户端已发送的某个请求。如果浏览器请求一个主体 HTML 页面，如果要推送此页面使用的某个 JavaScript 对象，服务器将使用请求对应的流 ID 构造 PUSH_PROMISE 帧。
- PUSH_PROMISE 帧的首部块与客户端请求推送对象时发送的首部块是相似的。所以客户端有办法放心检查将要发送的请求。
- 被发送的对象必须确保是可缓存的。
- `:method` 首部的值必须确保安全。安全的方法就是**幂等的**那些方法，这是一种不改变任何状态的好办法。例如，GET 请求被认为是幂等的，因为它通常只是获取对象，而 POST 请求被认为是非幂等的，因为它可能会改变服务器端的状态。
- 理想情况下，PUSH_PROMISE 帧应该更早发送，应当早于客户端接收到可能承载着推送对象的 DATA 帧。假设服务器要在发送 PUSH_PROMISE 之前发送完整的 HTML，那客户端可能在接收到 PUSH_PROMISE 之前已经发出了对这个资源的请求。h2 足够健壮，可以优雅地解决这类问题，但还是会有些浪费。
- PUSH_PROMISE 帧会指示将要发送的响应所使用的流 ID。

客户端会从 1 开始设置流 ID，之后每新开启一个流，就会增加 2，之后一直使用奇数。服务器开启在 PUSH_PROMISE 中标明的流时，设置的流 ID 从 2 开始，之后一直使用偶数。这种设计避免了客户端和服务器之间的流 ID 冲突，也可以轻松地判断哪些对象是由服务端推送的。0 是保留数字，用于连接级控制消息，不能用于创建新的流。

如果客户端对 PUSH_PROMISE 的任何元素不满意，就可以按照拒收原因选择重置这个流（使用 RST_STREAM），或者发送 PROTOCOL_ERROR（在 GOAWAY 帧中）。常见的情况是缓存中已经有了这个对象。[5] 而 PROTOCOL_ERROR 是专门留给 PUSH_PROMISE 涉及的协议层面问题的，比如方法不安全，或者当客户端已经在 SETTINGS 帧中表明自己不接受推送时，仍然进行了推送。值得注意的是，服务器可以在 PUSH_PROMISE 发送后立即启动推送流，因此拒收正在进行的推送可能仍然无法避免推送大量资源。推送正确的资源是不够的，还需要保证只推送正确的资源，这是重要的性能优化手段。

假设客户端不拒收推送，服务端会继续进行推送流程，用 PUSH_PROMISE 中指明 ID 对应的流来发送对象（如图 5-4 所示）。

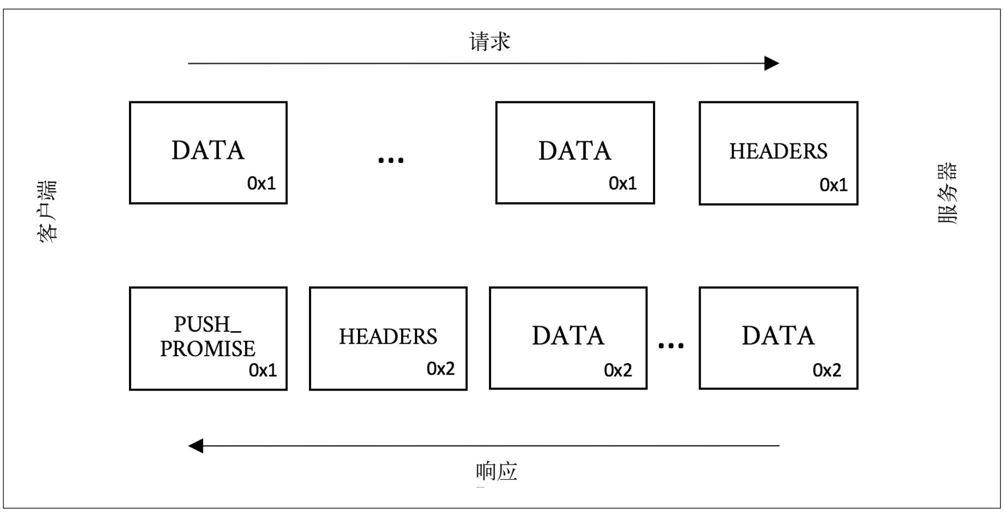


图 5-4：服务端推送消息处理

5.5.2 选择要推送的资源

根据应用的不同，选择推送哪些资源的逻辑可能非常简单，也可能异常复杂。拿一个简单的 HTML 页面来说，如果服务器接收到一个页面的请求，它需要决定是推送页面上的资源还是等客户端来请求。决策的过程需要考虑到如下方面：

注 5：此时会重置。——译者注

- 资源已经在浏览器缓存中的概率
- 从客户端看来，这些资源的优先级（参见 5.4.3 节）
- 可用的带宽，以及其他类似的会影响客户端接收推送的资源

如果服务器选择正确，那就真的有助于提升页面的整体性能，反之则会损耗页面性能。尽管 SPDY 早在 5 年前就已经引入了这个特性，但如今通用的服务端推送解决方案非常少见，原因可能就在这里。

更特殊的情况是，API 或通过 h2 通信的应用程序可能更容易决定近期需要什么，并知道客户端还没有缓存什么。想想服务端给原生应用推送的更新吧，这将是接下来在服务端推送上收益最大的场景。

5.6 首部压缩

3.1.3 节中“臃肿的消息首部”提到过，现代网页平均包含 140 个请求，每个 HTTP 请求平均有 460 字节，总数据量达到 63KB。即使在最好的环境下，这也会造成相当长的延时，如果考虑到拥挤的 WiFi 或连接不畅的蜂窝网络，那可是非常痛苦的。这些请求之间通常几乎没有新的或不同的内容，这才是真正的浪费。所以，大家迫切渴望某种类型的压缩。

一开始我们就知道，首部压缩（HPACK）将会是 HTTP/2 的关键元素之一。但是首部应该怎么压缩？浏览器的世界刚从 SPDY 的 CRIME 漏洞中恢复过来，该漏洞以创造性的方式利用 deflate 首部压缩算法来解密早期的加密帧，因此原有的方法肯定不行。我们需要的机制应当可以抵御 CRIME，同时具备和 GZIP 类似的压缩能力。

经过多次创新性的思考和讨论，人们提出了 HPACK。HPACK 是种表查找压缩方案，它利用霍夫曼编码获得接近 GZIP 的压缩率。要想了解 HPACK 的工作原理，最好的办法可能是举个简单例子。

> 为什么不直接用 GZIP 做首部压缩，而要使用 HPACK ？那样肯定能节省大量工作。不幸的是，CRIME 攻击告诉我们，GZIP 也有泄漏加密信息的风险。CRIME 的原理是这样的，攻击者在请求中添加数据，观察压缩加密后的数据量是否会小于预期。如果变小了，攻击者就知道注入的文本和请求中的其他内容（比如私有的会话 cookie）有重复。在很短的时间内，经过加密的数据内容就可以全部搞清楚。因此，大家放弃了已有的压缩方案，研发出 HPACK。

下载 Web 页面及其依赖的资源往往涉及大量的请求，单个 Web 页面的请求通常数以百计，而这些请求往往非常相似。以下面两个请求为例，它们看起来像是浏览器请求完整网页的会话中先后发生的。少数不同的字节用加粗字体强调。

第一个请求：

```
:authority: www.akamai.com
:method: GET
:path: /
:scheme: https
accept: text/html,application/xhtml+xml
accept-language: en-US,en;q=0.8
cookie: last_page=286A7F3DE
upgrade-insecure-requests: 1
user-agent: Awesome H2/1.0
```

第二个请求：

```
:authority: www.akamai.com
:method: GET
:path: /style.css
:scheme: https
accept: text/html,application/xhtml+xml
accept-language: en-US,en;q=0.8
cookie: last_page=*398AB8E8F
upgrade-insecure-requests: 1
user-agent: Awesome H2/1.0
```

可以看到，后者的很多数据与前者重复了。第一个请求约有 220 字节，第二个约有 230 字节，但二者只有 36 字节是不同的。如果仅仅发送这 36 字节，就可以节省约 85%的字节数。简而言之，HPACK 的原理就是这样。

下面是一个专门设计的简化的例子，来帮助你理解 HPACK 到底做了些什么。现实情况会更复杂，也没有那么理想，如果你想学习更多，应该阅读 RFC 7541，“HPACK：HTTP/2 的首部压缩”[6]。

假设客户端按顺序发送如下请求首部：

```
Header1: foo
Header2: bar
Header3: bat
```

当客户端发送请求时，可以在首部数据块中指示特定首部及其应该被索引的值。它会创建一张表：

索引	首部名称	值
62	Header1	foo
63	Header2	bar
64	Header3	bat

注 6：https://tools.ietf.org/html/rfc7541

如果服务端读到了这些请求首部，它会照样创建一张表。客户端发送下一个请求的时候，如果首部相同，它可以直接发送这样的首部块：

```
62 63 64
```

服务器会查找先前的表格，并把这些数字还原成索引对应的完整首部。

这里的首部压缩机制中每个连接都维护了自己的状态，这一点尤其值得注意，因为这在 h1 的协议层面中是不存在的。

HPACK 的实现比这个要复杂得多。读者若对此感兴趣，以下提供了一些线索。

- 实际上，请求端和响应端各维护了两张表格。其中之一是动态表，创建方法和上面差不多。另一张是静态表，它由 61 个最常见的首部的键值组合而成。例如 `:method: GET` 在静态表中索引为 2。按规定，静态表包含 61 个条目，所以上例索引编号从 62 开始。
- 关于字段如何索引，有很多控制规则，其中包含：
  - 发送索引编号和文本值（如上例所示）；
  - 仅发送文本值，不对它们进行索引（对于一次性或敏感首部）；
  - 发送索引的首部名，值用文本表示，但不进行索引处理（如 `:path: /foo.html`，其值每次都不同）；
  - 发送索引过的首部名和值（如上例中的第二个请求）。
- 使用打包方案的整数压缩，以实现极高的空间效率。
- 利用霍夫曼编码表进一步压缩字符串。

实验表明，HPACK 表现非常好，尤其是针对网站有大量重复首部（比如 cookie）的情况。由于到固定网站的各个请求的大部分首部信息是重复的，HPACK 的表查找机制有效去除了通信中的重复字节。

5.7 线上传输

下面来看一个 HTTP/2 的请求和响应，并逐层解析它。再强调一次，这里我们都用文本表示，是为了方便阅读，实际在线上传输的 h2 信息是经过压缩的二进制数据。

一个简单的GET请求

GET 是 HTTP 协议中的主力。它的语义简单，名副其实，用于从服务器获得一份资源。示例 5-1 是一个到 akamai.com 的请求（为清楚起见，部分较长的行已缩略）。

示例 5-1　HTTP/2 GET 请求

```
:authority: www.akamai.com
:method: GET
```

```
:path: /
:scheme: https
accept: text/html,application/xhtml+xml,...
accept-language: en-US,en;q=0.8
cookie: sidebar_collapsed=0; _mkto_trk=...
upgrade-insecure-requests: 1
user-agent: Mozilla/5.0 (Macintosh;...
```

这个请求通过 HTTPS 的 GET 方法，从 www.akamai.com 获取首页。其响应如示例 5-2 所示。

示例 5-1 中的首部名称 `:authority` 可能看起来有点奇怪。为什么不是 `:host` 呢？原因在于，它类似于 URI 中的 Authority 段，而不是 HTTP/1.1 中的 Host 首部。Authority 段包含主机信息，可能还有端口号，这样就刚好可以替代 Host 首部的角色。读过 URI RFC[7] 的读者需要注意，Authority 段里的 User Information（用户信息，如用户名和密码），在 h2 中是明令禁止的。

示例 5-2　HTTP/2 GET 响应（只包含首部信息）

```
:status: 200
cache-control: max-age=600
content-encoding: gzip
content-type: text/html;charset=UTF-8
date: Tue, 31 May 2016 23:38:47 GMT
etag: "08c024491eb772547850bf157abb6c430-gzip"
expires: Tue, 31 May 2016 23:48:47 GMT
link: <https://c.go-mpulse.net>;rel=preconnect
set-cookie: ak_bmsc=8DEA673F92AC...
vary: Accept-Encoding, User-Agent
x-akamai-transformed: 9c 237807 0 pmb=mRUM,1
x-frame-options: SAMEORIGIN

<DATA Frames follow here>
```

在这个响应中，服务器表示请求已成功受理（状态码 200），设置了 cookie（cookie 首部），表示返回的内容使用 gzip 压缩（content-encoding 首部），还发送了需要用到的其他重要信息。

先来看看这个简单的 GET 请求背后到底发生了什么。`nghttp`[8] 是 Tatsuhiro Tsujikawa 提供的强力工具，通过它可以看到详细信息的输出，并弄清楚 h2 的各个细节：

```
$ nghttp -v -n --no-dep -w 14 -a -H "Header1: Foo" https://www.akamai.com
```

这条命令把窗口大小设置为 16KB（2^{14}），添加了一个没有意义的首部，并请求下载页面的一些关键资源。下面是这个命令的详细输出，并加了注解：

注 7：https://www.ietf.org/rfc/rfc3986.txt

注 8：https://github.com/nghttp2/nghttp2

```
[ 0.047] Connected
The negotiated protocol: h2 ❶
[ 0.164] send SETTINGS frame <length=12, flags=0x00, stream_id=0> ❷
         (niv=2)
         [SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
         [SETTINGS_INITIAL_WINDOW_SIZE(0x04):16383] ❸
```

可以看到 `nghttp` 的情况如下。

❶ 成功协商建立 h2 连接。
❷ 按照规范，立即发送一个 SETTINGS 帧。
❸ 按命令行中的要求，将窗口大小设置为 16KB。

请注意，`stream_id 0` 用于连接层的信息。（你在输出中并没看到连接前奏，但它其实已经在 SETTINGS 帧之前发送过了。）

接下来是输出日志：

```
[  0.164] send HEADERS frame <length=45, flags=0x05, stream_id=1>
          ; END_STREAM | END_HEADERS ❹
          (padlen=0)
          ; Open new stream
          :method: GET
          :path: /
          :scheme: https
          :authority: www.akamai.com
          accept: */*
          accept-encoding: gzip, deflate
          user-agent: nghttp2/1.9.2
          header1: Foo ❺
```

这是请求的首部块。

❹ 注意，客户端（`nghttp`）发送了 END_HEADERS 和 END_STREAM 标识。这告诉服务器没有更多的首部，也没有其他数据了。如果这是 POST 请求，此时不会发送 END_STREAM 标识。
❺ 这是我们在 `nghttp` 命令行中添加的请求首部。

```
[  0.171] recv SETTINGS frame <length=30, flags=0x00, stream_id=0> ❻
          (niv=5)
          [SETTINGS_HEADER_TABLE_SIZE(0x01):4096]
          [SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
          [SETTINGS_INITIAL_WINDOW_SIZE(0x04):65535]
          [SETTINGS_MAX_FRAME_SIZE(0x05):16384]
          [SETTINGS_MAX_HEADER_LIST_SIZE(0x06):16384]
[  0.171] send SETTINGS frame <length=0, flags=0x01, stream_id=0> ❼
          ; ACK
          (niv=0)
```

```
[  0.197] recv SETTINGS frame <length=0, flags=0x01, stream_id=0>
          ; ACK
          (niv=0)
```

❻ nghttpd 收到了服务器的 SETTINGS 帧。

❼ 发送并接收到了 SETTINGS 帧的确认。

```
[  0.278] recv (stream_id=1, sensitive) :status: 200 ❽ ❾
[  0.279] recv (stream_id=1, sensitive) last-modified: Wed, 01 Jun 2016 ...
[  0.279] recv (stream_id=1, sensitive) content-type: text/html;charset=UTF-8
[  0.279] recv (stream_id=1, sensitive) etag: "0265cc232654508d14d13deb...gzip"
[  0.279] recv (stream_id=1, sensitive) x-frame-options: SAMEORIGIN
[  0.279] recv (stream_id=1, sensitive) vary: Accept-Encoding, User-Agent
[  0.279] recv (stream_id=1, sensitive) x-akamai-transformed: 9 - 0 pmb=mRUM,1
[  0.279] recv (stream_id=1, sensitive) content-encoding: gzip
[  0.279] recv (stream_id=1, sensitive) expires: Wed, 01 Jun 2016 22:01:01 GMT
[  0.279] recv (stream_id=1, sensitive) date: Wed, 01 Jun 2016 22:01:01 GMT
[  0.279] recv (stream_id=1, sensitive) set-cookie: ak_bmsc=70A833EB...
[  0.279] recv HEADERS frame <length=458, flags=0x04, stream_id=1> ❿
          ; END_HEADERS
          (padlen=0)
          ; First response header
```

现在拿到了服务端返回的响应首部。

❽ stream_id 为 1 表明响应对应的请求（我们刚刚只发了一个请求，但生活不会总如此简单）。

❾ nghttpd 从服务器获得了 200 状态码，这表示成功了。

❿ 注意，此时并没有发送 END_STREAM，因为下面还有 DATA 帧 。

```
[ 0.346] recv DATA frame <length=2771, flags=0x00, stream_id=1> ⓫
[ 0.346] recv DATA frame <length=4072, flags=0x00, stream_id=1>
[ 0.348] recv DATA frame <length=4072, flags=0x00, stream_id=1>
[ 0.348] send WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=1>
```

⓫ 最后我们从流里获得了数据。这里看到 3 个 DATA 帧，之后跟了一个 WINDOW_UPDATE 帧。客户端告诉服务器，它消耗掉了 10 915 字节的 DATA 帧，并为接下来更多的数据做好了准备。注意，此时流还没有结束，客户端还有其他事情要做，正好可以依靠多路复用。

```
[  0.348] send HEADERS frame <length=39, flags=0x25, stream_id=15> ⓬
         :path: /styles/screen.1462424759000.css
[  0.348] send HEADERS frame <length=31, flags=0x25, stream_id=17>
         :path: /styles/fonts--full.css
[  0.348] send HEADERS frame <length=45, flags=0x25, stream_id=19>
         :path: /images/favicons/favicon.ico?v=XBBK2PxW74
```

⓬ 客户端已经得到了主体 HTML 的部分内容，现在可以请求页面中的资源了。现在你看到 3 个新建的流，ID 分别是 15、17 和 19，其中有两个用于 CSS，一个用于 favicon。（为了方便读者理解，这里跳过和简化了一些帧。）

```
[ 0.378] recv DATA frame <length=2676, flags=0x00, stream_id=1>
[ 0.378] recv DATA frame <length=4072, flags=0x00, stream_id=1>
[ 0.378] recv DATA frame <length=1445, flags=0x00, stream_id=1>
[ 0.378] send WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=13>
         (window_size_increment=12216)
[ 0.379] recv HEADERS frame <length=164, flags=0x04, stream_id=17> ⓭
[ 0.379] recv DATA frame <length=175, flags=0x00, stream_id=17>
[ 0.379] recv DATA frame <length=0, flags=0x01, stream_id=17>
         ; END_STREAM
[ 0.380] recv DATA frame <length=2627, flags=0x00, stream_id=1>
[ 0.380] recv DATA frame <length=95, flags=0x00, stream_id=1>
[ 0.385] recv HEADERS frame <length=170, flags=0x04, stream_id=19> ⓭
[ 0.387] recv DATA frame <length=1615, flags=0x00, stream_id=19>
[ 0.387] recv DATA frame <length=0, flags=0x01, stream_id=19>
         ; END_STREAM
[ 0.389] recv HEADERS frame <length=166, flags=0x04, stream_id=15> ⓭
[ 0.390] recv DATA frame <length=2954, flags=0x00, stream_id=15>
[ 0.390] recv DATA frame <length=1213, flags=0x00, stream_id=15>
[ 0.390] send WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=0>
         (window_size_increment=36114)
[ 0.390] send WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=15> ⓮
         (window_size_increment=11098)
[ 0.410] recv DATA frame <length=3977,  flags=0x00, stream_id=1>
[ 0.410] recv DATA frame <length=4072,  flags=0x00, stream_id=1>
[ 0.410] recv DATA frame <length=1589,  flags=0x00, stream_id=1> ⓯
[ 0.410] recv DATA frame <length=0, flags=0x01, stream_id=1>
[ 0.410] recv DATA frame <length=0, flags=0x01, stream_id=15>
```

此时可以看到服务端发过来的流交织在一起。

⓭ 你可以看到 ID 为 15、17 和 19 的 HEADERS 帧。

⓮ 这些对应不同的窗口更新，包含一个连接层的更新，流 ID 为 0。

⓯ ID 为 1 的流最后的 DATA 帧。

```
[ 0.457] send GOAWAY frame <length=8, flags=0x00, stream_id=0>
         (last_stream_id=0, error_code=NO_ERROR(0x00), opaque_data(0)=[])
```

最后我们看到了 GOAWAY 帧。虽然取了这么个名字，它却是断开连接的礼貌方式。[9]

这个过程乍一看可能有点神秘，但是多试几次就会熟悉了。从头到尾，一切都遵从逻辑、符合规范、用途明确。在这个简单的例子中，你可以看到构成 h2 的许多元素，包括流量控制、多路复用，以及连接设置。你可以用 `nghttp` 工具多测试一些支持 h2 的网站，看看是否可以走通上面的流程。熟练之后，你就已经迈过了理解协议的门槛。

注 9：英文 go away 的意思是“滚开”。——译者注

5.8 小结

HTTP/2 协议的制定经过了许多年，包含了各种设计理念、决策、创新，以及妥协。本章提供了一些基础知识，让读者能够查看 h2 的 Wireshark 导出文件（参见 8.7 节），并了解背后的原理，甚至可以帮助读者在实际使用 HTTP/2 的时候发现潜在问题（也许是不断更改 cookie）。对想要深入研究的读者来说，最好的学习资源就是 RFC 7540 本身[10]。无论是实现者、调试者，还是你内心潜伏的受虐狂人格，需要的所有细节都在这里。

注 10：https://tools.ietf.org/html/rfc7540

第 6 章

HTTP/2性能

HTTP/2 的设计目标之一就是提升性能。对于某些人来说，这是迁移的唯一理由。虽然 h2 大部分情况下传输 Web 页面比 HTTP/1.1 快，但并非总是如此。因此，要进行网站调优，并了解最终用户体验，理解影响 h2 性能的因素就很重要。在衡量 h2 的性能时需要考虑的因素很多，评价也不能简单分成“更快”和“更慢”两类。因此，本章将会探索有助于在真实世界中提升 HTTP/2 性能的关键因素。

6.1 客户端实现

HTTP/2 的协议历史不长，所以和其他新鲜事物一样，尽管各个实现版本与规范兼容，却存在微小而重要的差异。例如，像 Chrome、Firefox、Safari 和微软 Edge 这样的浏览器都有各自的特殊表现，可能会影响网站的用户体验。假设网络条件相同，使用不同客户端，同样的网站页面加载性能可能差别迥异。所以，要针对尽可能多的用户来调优网站性能，关键就在于理解那些差异。

Akamai 的 Foundry 团队做过一份 HTTP/1.1 与 HTTP/2 性能对比的研究，目标是弄清楚不同浏览器的差异。这项研究包括查看数十亿的真实用户监控（RUM）测量结果，这些数据均来自于真实网络条件下的真实设备。结果显示了浏览器间的差异，而且对大部分页面而言，h2 请求比 h1 请求更快（见图 6-1）。

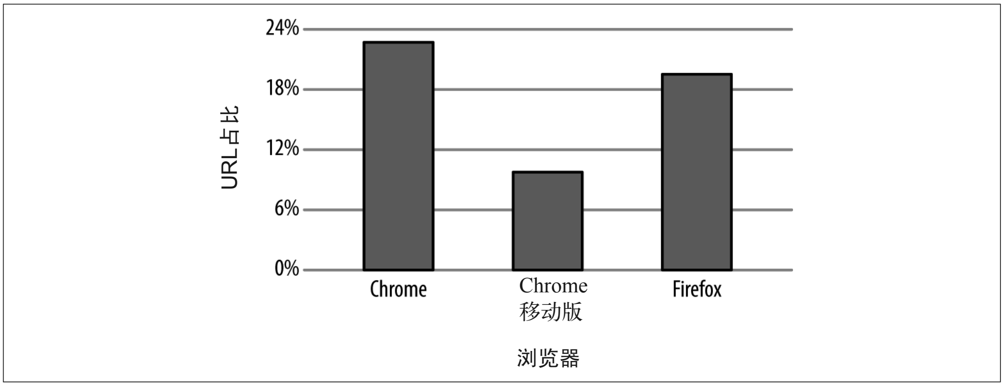


图 6-1：使用 HTTP/2 之后性能得到显著提升的 URL 占比（差别具有统计学意义）

任务完成了，对吗？其实没那么简单。这项研究同样表明，开启 h2 时，某些 URL 的性能会下降（见图 6-2）。

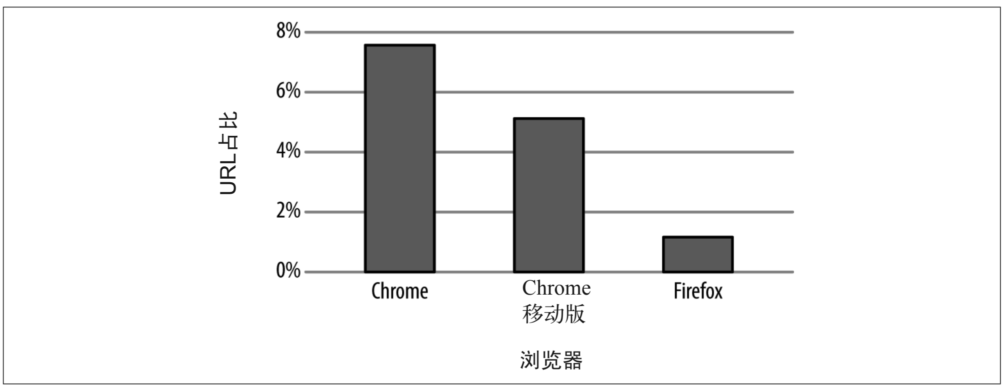


图 6-2：使用 HTTP/2 之后性能显著下降的 URL 占比（差别具有统计学意义）

为什么用了 h2 反而更慢？因为这项研究关注的是单个请求，而不是整个页面的加载，理解这一点很重要。这就意味着，仅有的提升空间在于首部压缩、连接重用和避免队头阻塞。像多路复用和服务端推送（Server Push）这样的技术，更关注于如何提升页面上多个请求的性能，因此对于这项研究的结果没有影响。即便如此，URL 使用 h2 后性能提升的比例也依旧高于下降的比例。数据的差异凸显了两个重点：第一，协议的具体实现很重要；第二，并非所有请求在任何情况下都会从 HTTP/2 受益。

6.2 延迟

在计算机网络中，延迟是指数据包从一个端点到另一个端点所花的时间。有时，它也表示数据包到达接收方然后返回发送方所需的时间，又称为往返时延（RTT），长度一般以毫秒计。

虽然影响延迟的因素众多，但有两个是最重要的：端点间的距离，以及所用传输介质。在

有线网络中，传输介质一般由光纤或铜丝制成，而移动 / 无线网络则利用无线电波来传输信号。端点间的理论最小延迟，取决于光在介质中传播的速度以及传输线路的长度。例如，光在光纤中传播的速度大约是在真空中的 2/3，相当于每秒 20 万千米。所以，如果直接从旧金山拉一根光纤到伦敦，距离约 8500 千米，其最小可能延迟大概是 43 毫秒。减少延迟时间的唯一方式就是让两端靠得更近（或者研发更快的传输介质）。

无论传输介质多么先进，光速始终是无法突破的极限。因此，让两端靠得更近是最有可能减少延迟的办法。开个玩笑，大陆板块漂移迟早会解决“旧金山－伦敦”之间的距离问题，但是这可能需要数百万年之久，等不及的读者或许可以把服务器部署得更靠近世界各地的最终用户，或者借助 CDN 来达到这一目的（参见 7.5 节）。

当然，如果你自己动手检测，就会发现事实上延迟时间会更长。这是因为：第一，两点之间的网线不会是笔直的；第二，各种网关、路由器、交换机以及移动基站等（也包括服务器应用本身）都会增加延迟，数据从一端到达另一端必须经过这些设施。

增加带宽不会减少延迟

2010 年，Mike Belshe（SPDY 协议的共同发明人）发表了题为“More Bandwidth Doesn’t Matter (Much)”[1] 的研究，主题是带宽与往返时延（RTT）对 Web 页面加载时间的影响。

他的测试表明，增加带宽与减少 Web 页面下载时间是相关的。尽管如此，一旦带宽达到 5Mbit/s，性能提升幅度就会降低，在大约 8Mbit/s 或更高时几乎停滞。另一方面，如果 RTT 减少，页面加载时间（PLT）却呈指数级下降（这项研究中，如果每隔 20 毫秒减少一次 RTT，那么 PLT 会下降 7%~15%）。

简而言之，只要减少 RTT，不管当前带宽是多少，总会对加速 Web 浏览有帮助。

图 6-3（摘自该研究）说明了带宽与 RTT 对页面加载时间的影响。

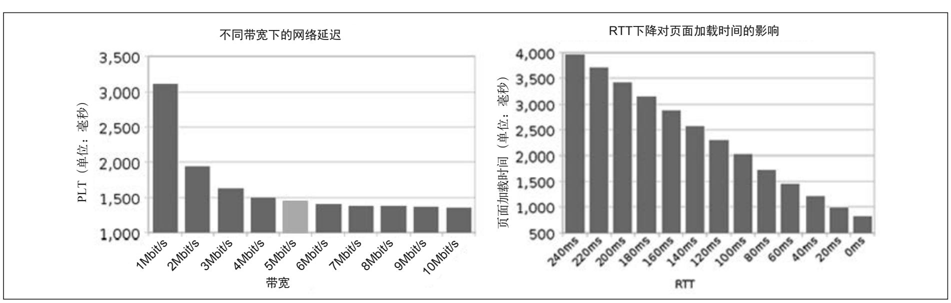


图 6-3：带宽与 RTT 对 PLT 的影响

注 1：https://docs.google.com/a/chromium.org/viewer?a=v&pid=sites&srcid=Y2hyb21pdW0ub3JnfGRldnxneDoxMzcyOWI1N2I4YzI3NzE2

你可以简单地通过 ping 命令工具测量客户端与服务器之间的延迟，大多数操作系统中都包含这个工具。

下面使用 ping 命令测量维基百科网站 RTT 的输出：

```
$ ping -c 4 www.wikipedia.org
PING www.wikipedia.org (208.80.154.224) 56(84) bytes of data.
64 bytes from text-lb.eqiad.wikimedia.org (...): icmp_req=1 ttl=50 time=70.4 ms
64 bytes from text-lb.eqiad.wikimedia.org (...): icmp_req=2 ttl=50 time=70.7 ms
64 bytes from text-lb.eqiad.wikimedia.org (...): icmp_req=3 ttl=50 time=70.5 ms
64 bytes from text-lb.eqiad.wikimedia.org (...): icmp_req=4 ttl=50 time=70.5 ms

--- www.wikipedia.org ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3002ms
rtt min/avg/max/mdev = 70.492/70.571/70.741/0.284 ms
```

这条 ping 命令是在位于加州圣何塞的客户端上运行。根据地理位置数据，维基百科的主机服务器 IP 地址 208.80.154.224 位于弗吉尼亚州阿什本（距离加州的客户端大约 3850 千米）。

表 6-1 展示的是平均延迟采样，其中的值依赖于传输介质。

表6-1：常见传输介质的延迟值

介质类型	平均RTT
光纤	17~22 毫秒
有线网络	15~30 毫秒
DSL	32~52 毫秒
移动网络	40~1000 毫秒，取决于所用的无线网络技术，具体有 LTE（最快）、HSPA，以及 GSM/Edge（最慢）等
卫星	600~650 毫秒

请注意，一些移动设备为节省电力，可能暂时关闭移动数据信号。如果设备需要临时唤醒移动数据设备，建立新连接时还要增加数秒的延迟。

为了测量 h2 中延迟的影响，我们建了个简单的网站，从页面大小和包含资源数来看，它可以代表排名前 1000 位的某网站的一个普通页面。然后，我们在站点前设置了 CDN（参见 7.5 节），以便能从全球各地访问来获得各种“真实”延迟的数据。我们使用基于 Web 的免费性能测试工具 WebPagetest（缩写为 WPT，参见 8.2 节），分别基于 h1 和 h2 在 Chrome 和 Firefox 中加载该页面。

表 6-2 显示了 h1 和 h2 中延迟对页面加载时间的影响。PLT 时间是两天内重复测试 20 次的结果取平均值，每次测试包含 9 个“首屏”WPT 测试用例。

表6-2：h1与h2在真实延迟下的性能对比，使用WPT的客户端代理：选择地点为弗吉尼亚州杜勒斯，采用有线连接

源所在地	延迟	h1的PLT Chrome（单位：毫秒）	h2的PLT Chrome（单位：毫秒）	h1的PLT Firefox（单位：毫秒）	h2的PLT Firefox（单位：毫秒）
美国，纽约	15 毫秒	4518	5064	4673	4637
加拿大，蒙特利尔	39 毫秒	4719	5325	4815	4718
美国，得克萨斯州达拉斯	42 毫秒	4728	4986	4965	4995
法国，巴黎	97 毫秒	6248	5765	5634	5402
埃及，开罗	129 毫秒	6873	5272	5266	5256
巴西，里约热内卢	142 毫秒	7302	5932	6055	6220

从以上数据很容易总结出一条规律：总体来说，延迟会随着到源端点的距离的增加而增加，但 h2 的延迟增长低于 h1。

6.3　丢包

如果网络中传输的数据包没有成功到达目的地，就会发生丢包；这通常是由网络拥堵造成的。丢包通过丢包总数与已发送包总数的比值来衡量。频繁丢包会影响 h2 的页面传输，主要是因为 h2 开启单一 TCP 连接，每次有丢包 / 拥堵时，TCP 协议就会缩减 TCP 窗口（参见 3.1.3 节的“低效的 TCP 利用”）。

关于蜂窝网络下 HTTP/2 的性能，最近一项由蒙大拿大学与 Akamai 公司 Foundry 团队合作进行的研究[2]，分析了丢包对不同内容类型的影响（主要是指资源大小）。

该项研究有如下发现。

- 对于包含很多小型资源（365 个，每个 2KB）的 Web 页面，h2 加载页面的时间比 h1 更短。这是因为 h1 下（有 6 个 TCP 连接）服务器只能并行发送 6 个资源（由于队头阻塞），而 h2 下多个流（stream）可以共用连接。进一步说，随着网络条件变差，h1 和 h2 下 PLT 都会增加；但是对 h2 的影响更值得注意，因为这是单连接架构造成的。如果唯一的连接发生了丢包，所有工作都会受影响（见图 6-4a）。
- 对于包含少量大型资源的 Web 页面（10 个，每个 435KB），在所有网络条件下，h1 性能上都比 h2 表现要好。这个多少令人感到意外的结果是初始拥塞窗口（见图 3-4）导致的。如果开启 6 个连接，h1 的初始拥塞窗口大小实际上是 h2 的 6 倍。这意味着，在会话开始阶段，h2 的连接窗口尚未增长到最佳值，但 h1 早就能更快地传输更多数据了。这个问题目前仍在解决，因为它导致初始拥塞窗口对 h2 而言太小，然而对 h1 而言又太大。此外，h2 比 h1 更容易受丢包的影响（见图 6-4b）。

注 2：https://www.akamai.com/us/en/multimedia/documents/technical-publication/http2-performance-in-cellular-networks.pdf

- 对于包含一些极大型资源的 Web 页面，两者没有任何差异。h2 的初始拥塞窗口劣势被整体下载时长掩盖了，多路复用此时也不再具有优势。

大部分 Web 页面属于第一种情况（包含很多小资源），这时 h2 的优势最大。这并不是巧合，因为它确实是设计者之前努力优化的使用场景。不管怎么说，丢包是 h2 的命门。

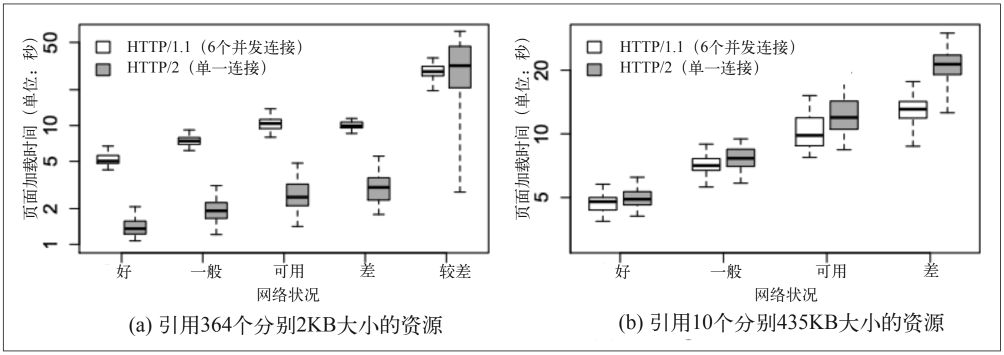


图 6-4：蜂窝网络下 HTTP/2 的性能

HTTP/2 的性能体现在何处

虽然 WebPagetest 是一个很棒的工具（参见 8.2 节），但是它缺少用简单的方式获得足量统计数据的能力。在网站上跑少量的测试能给你大致的感觉，告诉你哪里可能有问题；但如果是小规模样本上 10%~20% 的性能提升，我们仍然很难回答“我的网站变快了吗”这样的问题。像 Catchpoint、Keynote 和 Gomez 这样的工具可以让你观察更多的测试。相比先前狭义的测试，这肯定是不小的进步。但是，它们仍然是所谓的**模拟测试**，反映的不是真正的网站流量。

其实你想知道的是用户的真实体验，于是就有了真实用户监控（RUM）。它是性能统计数据收集的标准。你可以考虑用工具（例如 Boomerang）自己动手采集，或者联系像 SOASTA 或 Speedcurve 这样的公司。无论哪种方式，你都会获得 RUM 监测结果，那些数据是一座等待被好好挖掘的金矿。

一旦你拥有了那些数据，请以百分比而非平均值为单位进行分析。没有哪个用户是平均用户，但是他们中一些人确实体验会高于或低于平均值。分析中位数，你会知道有 50% 的用户的体验比这更好（或者更差）。再来看看 95%，甚至 99% 的百分比，这会显示用户正经历的一些最糟糕的体验。以这种方式来看性能，可以定位到特定的用户群，监控你的变更对他们的影响。

那么，h2 到底表现如何？我们通常会看到性能中位数的提升，以及在 95% 及以上级别的更大提升。更重要的是，95% 级别上的性能提升也许决定了用户是留下来继续使用你的网站，还是放弃并转向竞争对手的网站。聚焦于网站性能的短板往往会产生更大的业务价值。这等于说，h2 的口号也可以是“让你的网站不那么糟糕”。问题是，这个口号不怎么吸引人。

6.4 服务端推送

之前 5.5 节中讨论过，服务端推送让服务器具备了在客户端请求之前就推送资源的能力。测试表明，如果合理使用推送，页面渲染时间可以减少 20%~50%。

尽管如此，推送也会浪费带宽，这是因为服务端可能试图推送那些在客户端已经缓存的资源，导致客户端收到并不需要的数据。客户端确实可以发送 RST_STREAM 帧来拒绝服务器的 PUSH_PROMISE 帧，但是 RST_STREAM 并不会即刻到达，所以服务器还是会发送一些多余的信息。

如果用户第一次访问页面时，就能向客户端推送页面渲染所需的关键 CSS 和 JS 资源，那么服务端推送的真正价值就实现了。不过，这要求服务器端实现足够智能，以避免“推送承诺”（push promise）与主体 HTML 页面传输竞争带宽。理想情况下，服务端正在处理 HTML 页面主体请求时才会发起推送。有时候，服务端需要做一些后台工作来生成 HTML 页面。这时候服务端在忙，客户端却在等待，这正是开始向客户端推送所需资源的绝佳时机。图 6-5 说明了这样安排来提升性能的原理。

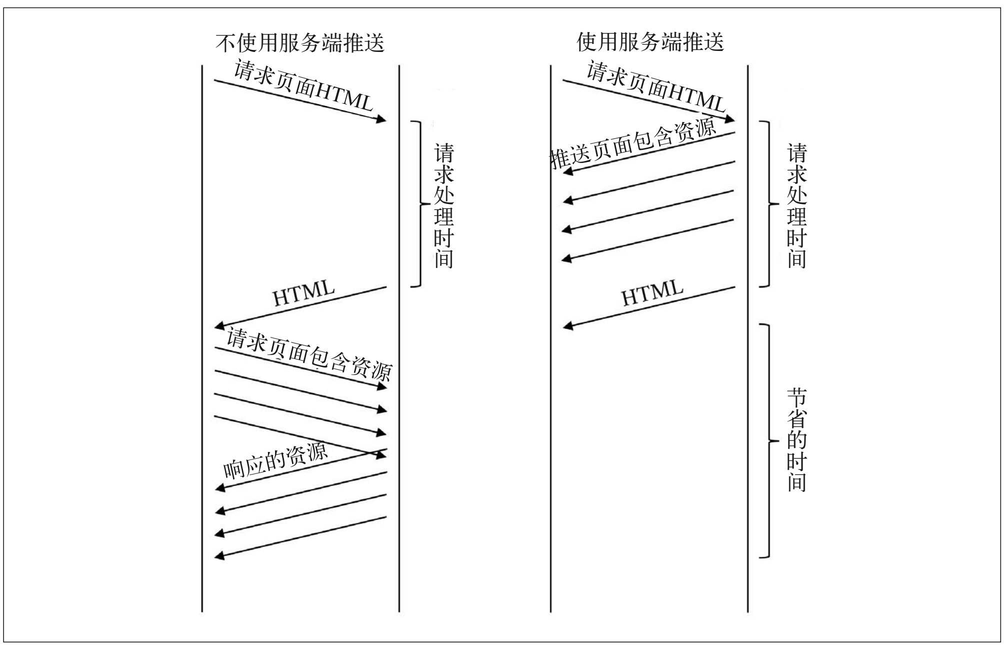


图 6-5：在后台处理的同时进行推送

我的客户端和服务器支持服务端推送吗

2016 年，在阿姆斯特丹举行的 Velocity 会议中，Colin Bendell 在演讲中分享了一个简单实用的网站 canipush.com。它允许你检查自己的客户端是否支持服务端推送。

表 6-3 列出了一些使用目前最流行的浏览器访问该网站时看到的输出。

表6-3：浏览器对服务端推送的支持

浏览器	Chrome 54.0	Firefox 50.0.1	Safari iOS 10.1	Edge
启用 HTTP/2	成功	成功	成功	成功
当前域名推送 JavaScript	成功	成功	成功	成功
当前域名推送 CSS	成功	成功	成功	成功
当前域名推送 XHR	成功	成功	成功	失败
针对 JavaScript 请求的连接归并	成功	成功	失败	失败
针对 CSS 请求的连接归并	成功	成功	失败	失败
针对 XHR 请求的连接归并	失败	失败	失败	失败

6.5 首字节时间

首字节时间（TTFB）用于测量服务器的响应能力。

TTFB 测量的是从客户端发起 HTTP 请求到客户端浏览器收到资源的第一个字节所经历的时间。它由 socket 连接时间、发送 HTTP 请求所需时间、收到页面第一个字节所需时间组成。虽然有时它被误解为忽略了 DNS 解析的时间，但是网络术语中，TTFB 的原始计算方法都包括了收到第一个字节之前的所有网络延迟。

在 h1 中，客户端针对单个域名在每个连接上依次请求资源，而且服务器会按序发送这些资源。客户端只有接收了之前请求的资源，才会再请求剩下的资源；服务器接着继续响应新的资源请求。这个过程会一直重复，直到客户端接收完渲染页面所需的全部资源。

与 h1 不同，通过 h2 的多路复用，客户端一旦加载了 HTML，就会向服务器并行发送大量请求。相比 h1，这些请求获得响应的时间之和一般会更短；但是因为是请求是同时发出的，而单个请求的计时起点更早，所以 h2 统计到的 TTFB 值反而更高。既然 h1 与 h2 协议的运行机制不同，TTFB 的意义也会随之变化。

HTTP/2 比 h1 确实做了更多的工作，其目的就是为了从总体上提升性能。下面是一些 h1 没有，但 h2 实现了的事情：

- 窗口大小调节
- 依赖树构建
- 维持首部信息的静态 / 动态表
- 压缩 / 解压缩首部
- 优先级调整（h2 允许客户端多次调整单一请求的优先级）
- 预先推送客户端尚未请求的数据流

图 6-6 和图 6-7 凸显了 h2 相比 h1 的优势。为了生成图中展示的结果，我们基于 h1 和 h2 加载同一页面。虽然在 TTFB 和“页面标题显示时间”（Time to Title）之类的指标上，两者可能表现旗鼓相当，或者 h1 更胜一筹，但是总体来说 h2 体验更好。

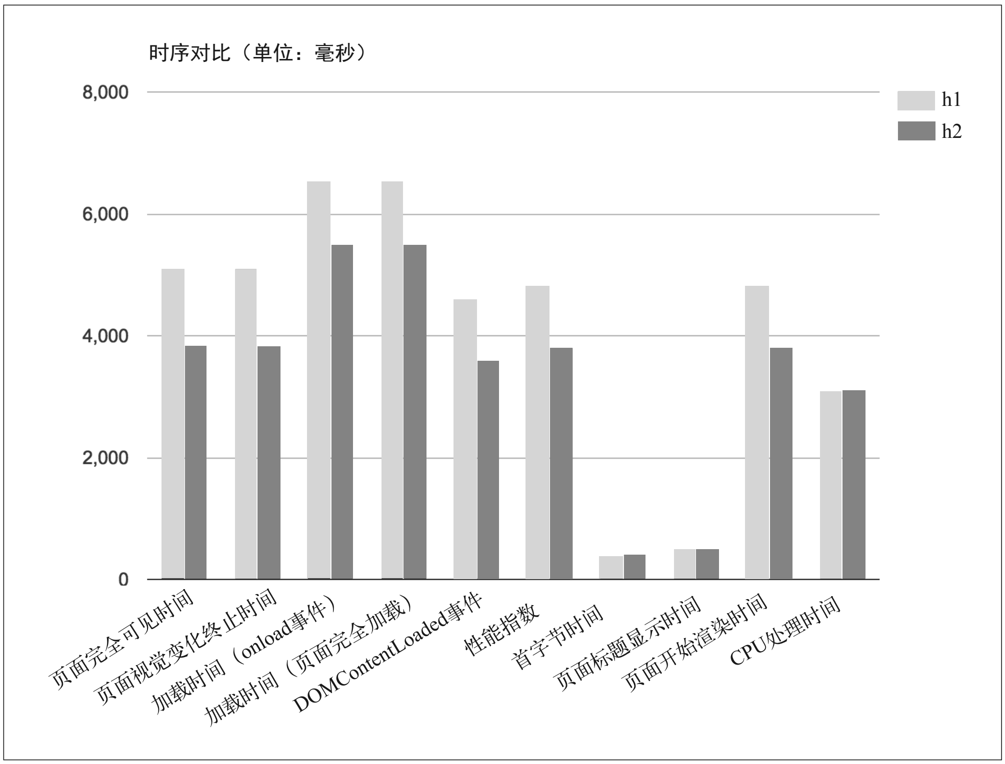


图 6-6：h1 与 h2 请求的时间序列



图 6-7：h1 与 h2 请求页面的渲染速度

6.6　第三方资源

如今，许多网站会使用各种统计、跟踪、社交以及广告平台，使用它们必须引入名为“第三方标签”的代码，并在网站上引用这些代码。第三方标签是外部供应商提供的代码，运行它们将发起指向第三方服务器的请求。

大量研究表明，第三方调用会让网站变慢，在阻塞 JavaScript 的情况下甚至会导致网站失去响应。包含大量第三方内容的页面，其性能在 h2 上并没有太大改观。第三方内容一般会降低网站性能，尤其对 h2 有以下几点特别影响。

- 第三方请求往往通过不同域名发送；由于浏览器需要解析 DNS、建立 TCP 连接、协商 TLS，这将严重影响性能。
- 因为第三方资源在不同域名下，所以请求不能从服务端推送、资源依赖、请求优先级等 h2 特性中受益。这些特性仅是为请求相同域名下的资源设计的。
- 你无法控制第三方资源的性能，也无法决定它们是否会通过 h2 传输。

换个角度来看，如果第三方内容占了页面加载时间的一半，那么 h2 只能解决一半的性能问题。

在 Web 页面语境下，单点故障（SPOF）是指 Web 页面上引用的某个资源，如果它出问题，将延迟整个页面的加载（甚至导致页面出错）。Pat Meenan 是一位软件工程师兼性能专家，因致力于建设 WebPagetest 平台而著称，他写了一个非常有用的 Chrome 浏览器插件 SPOF-O-MATIC[3]。有了这个插件，浏览网页时很容易就能检测出 SPOF 问题，并且通过使用 WebPagetest 来图形化展示那些问题的影响。它也是一个很棒的诊断工具，你应该将把它收入囊中。

注 3：https://github.com/pmeenan/spof-o-matic

为了测试第三方调用对性能的影响，我们准备了 4 个简单的 HTML 页面，每个页面包含 5 张来自指定域名的图片。HTML 代码大致如下：

```
<html>
  <head lang="en">
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>What is your Ikigai?</title>
  </head>
  <body>
    <img src="https://akah1san.h2book.com/what.png">
    <img src="https://akah1san.h2book.com/is.png">
    <img src="https://akah1san.h2book.com/your.png">
    <img src="https://akah1san.h2book.com/ikigai.png">
    <img src="https://akah1san.h2book.com/question-mark.png">
  </body>
</html>
```

我们通过改变图片所在域名来控制请求使用 h1 还是 h2。组合下列参数创建测试用例：

- 主体页面是基于 h2 还是 h1；
- 资源是基于 h2 还是 h1；
- 资源所在的域名与主体页面是否在同一证书下 [如果是，将允许连接归并（connection coalescing）]。

因为资源数量少，所以结果基本没什么区别。不过我们注意到，使用 h2 时，图片会早 100 毫秒开始显现出来，这是因为仅仅需要开启一个连接，如图 6-8 所示。到服务器的延迟越久，这种节省就越可观。

如果保持每个域名下资源个数相同并增加到 4 个拆分域名，我们能更清楚地看到 h1 开启更多连接对性能的影响。接下来，把前面例子中的 HTML 修改成重复引用 5 张图片，并把那些图片放在额外的域名下（注意，4 个域名均隶属于同样的 SAN 证书）：

```
<html>
  <head lang="en">
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>What is your Ikigai?</title>
  </head>
  <body>
    <img src="https://akah2san.h2book.com/what.png">
    <img src="https://akah2san.h2book.com/is.png">
    <img src="https://akah2san.h2book.com/your.png">
    <img src="https://akah2san.h2book.com/ikigai.png">
    <img src="https://akah2san.h2book.com/question-mark.png">
    <img src="https://akah2san1.h2book.com/what.png">
    <img src="https://akah2san1.h2book.com/is.png">
    <img src="https://akah2san1.h2book.com/your.png">
    <img src="https://akah2san1.h2book.com/ikigai.png">
    <img src="https://akah2san1.h2book.com/question-mark.png">
```

```
    <img src="https://akah2san2.h2book.com/what.png">
    <img src="https://akah2san2.h2book.com/is.png">
    <img src="https://akah2san2.h2book.com/your.png">
    <img src="https://akah2san2.h2book.com/ikigai.png">
    <img src="https://akah2san2.h2book.com/question-mark.png">
    <img src="https://akah2san3.h2book.com/what.png">
    <img src="https://akah2san3.h2book.com/is.png">
    <img src="https://akah2san3.h2book.com/your.png">
    <img src="https://akah2san3.h2book.com/ikigai.png">
    <img src="https://akah2san3.h2book.com/question-mark.png">
  </body>
</html>
```

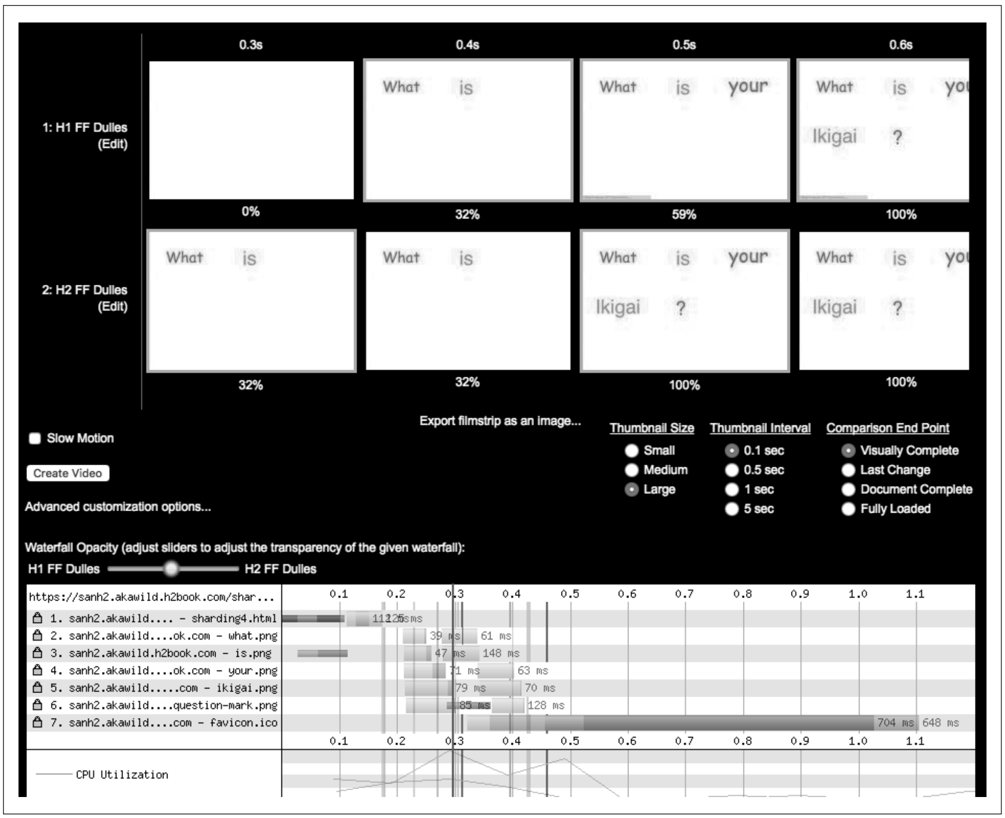


图 6-8：h1 与 h2 下加载页面产生的 WPT 截屏和时间线，该页面的资源分布在 2 个域名下

图 6-9 显示了加载该页面的结果。可以看到，通过 h2 发送页面时，加载速度快了约 25%。

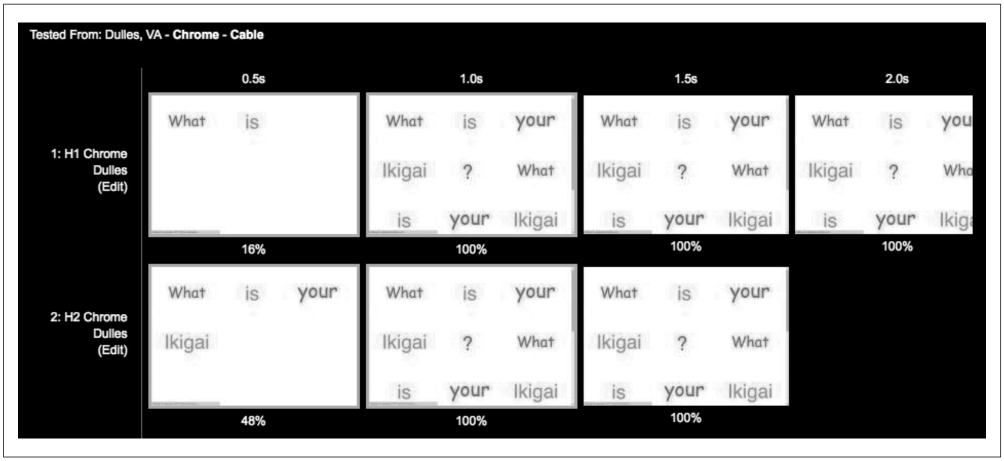


图 6-9：WPT 截屏：分别在 h1 和 h2 下加载使用 4 个拆分域名的页面

如果我们深入观察通过 h1 加载页面生成的时间线（见图 6-10），就会看到大多数加载引用图片请求的开始阶段都用于执行连接初始化和 SSL 握手。这显示了 h1 下开启多个连接的代价。

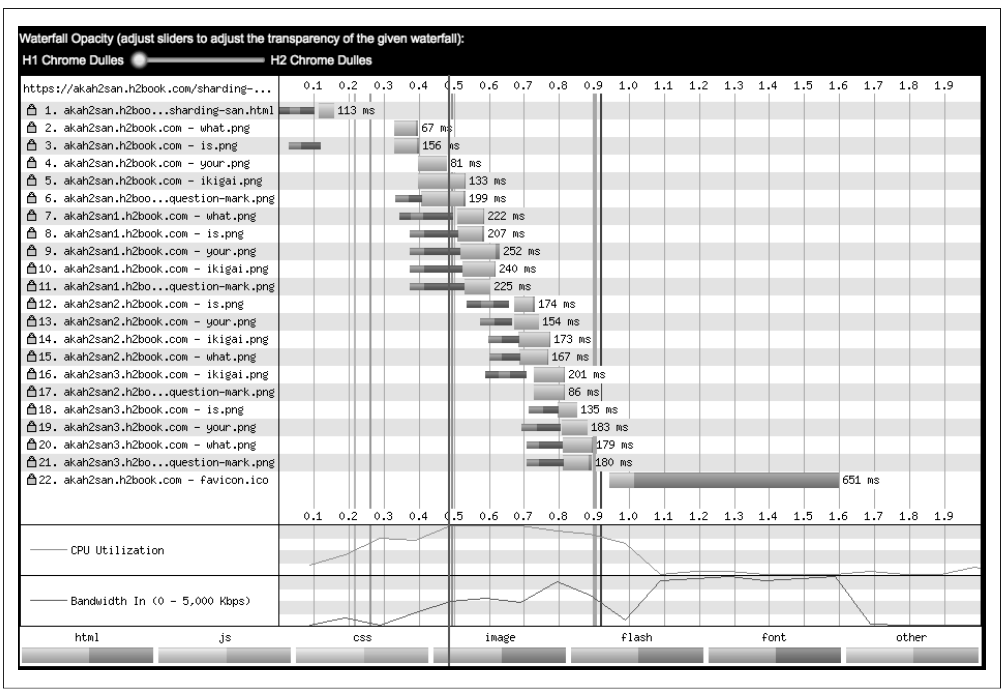


图 6-10：WPT 截屏：比较 h1 下加载使用 4 个拆分域名的页面

相反，如果页面通过 h2 加载（见图 6-11），只有第一份资源需要花时间建立连接和 TLS，剩下的资源也会通过这个连接发送。

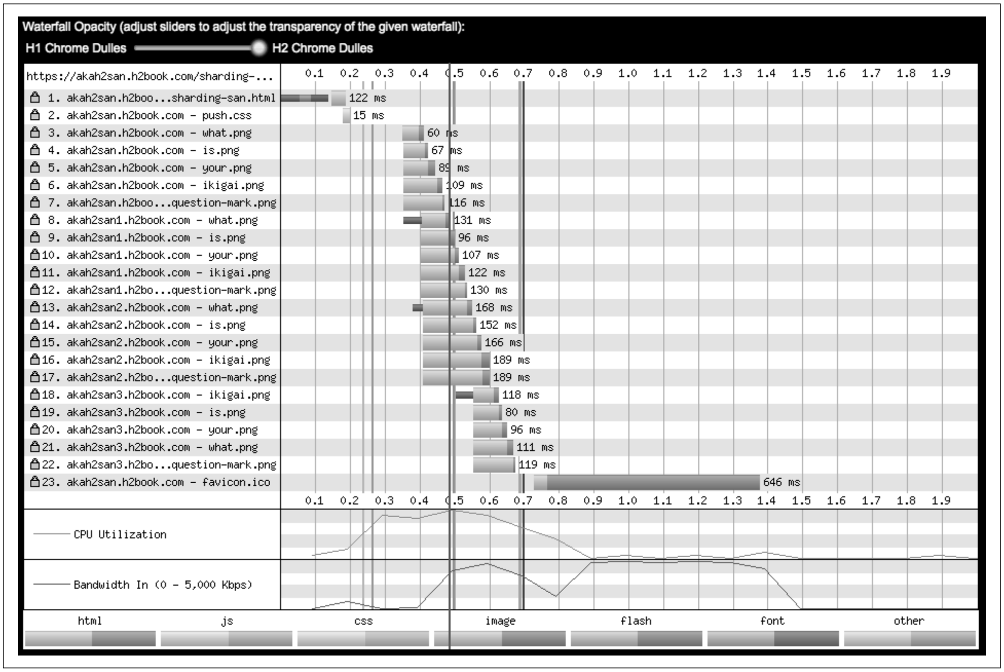


图 6-11：WPT 截屏：比较 h2 下加载使用 4 个拆分域名的页面

本节的实验表明，把做过域名拆分的网站迁移到 h2，并通过各种方式把不同域名归拢到同一份证书下，从浏览器（比如 Chrome 和 Firefox）实现的**连接归并**（connection coalescence）中受益，以便达到单一连接的效果，这种办法是可行的。

6.7 HTTP/2反模式

之前 4.3 节中已经说过，h1 下的一些性能调优办法在 h2 下会起到反作用。本节将回顾变通方法的共同之处，并详细描述在 h2 中它们对性能的影响。

6.7.1 域名拆分

域名拆分是为了利用浏览器对每个域名开启多个连接的能力，以便实现资源的并行下载，绕过 h1 的串行化下载的限制。对于包含大量小型资源的网站，普遍的做法是拆分域名，以利用现代浏览器针能对每个域名开启 6 个连接的特性。这样实际上做到了让浏览器并行发送多个请求，以及充分利用可用带宽的效果。因为 HTTP/2 采取多路复用，所以域名拆分就不是必要的了，并且反而会让协议力图实现的目标落空。在某些场景下，连接归并

（参见 7.1.4 节）可能会在部分浏览器中消除域名拆分带来的不利影响，但是最好不要依赖于它，而应该在 h2 中完全避免域名拆分。

6.7.2　资源内联

资源内联包括把 JavaScript、样式，甚至图片插入到 HTML 页面中，目的是省掉加载外部资源所需的新连接以及请求响应的时间。然而，有些 Web 性能的最佳实践不推荐使用内联，因为这样会损失更有价值的特性，比如缓存。如果有同一个页面上的重复访问，缓存通常可以减少请求数（而且能够加速页面渲染）。尽管如此，总体来说，对那些渲染滚动条以上区域所需的微小资源进行内联处理仍是值得的。事实上有证据表明，在性能较弱的设备上，缓存对象的好处不够多，把内联资源拆分出来并不划算。

使用 h2 时的一般原则是避免内联，但是内联也并不一定毫无价值（更多信息参见下面的附注栏“性能优化因人而异”）。

6.7.3　资源合并

资源合并意味着把几个小文件合并成一个大文件。它与内联很相似，旨在省掉那些加载外部资源的请求响应时间，以及解码 / 执行那些资源所消耗的 CPU 资源。之前针对资源内联的规则同样适用于资源合并，我们可以使用它来合并非常小的文件（1KB 或更小），以及对初始渲染很关键的最简化 JavaScript/CSS 资源。

6.7.4　禁用cookie的域名

通过禁用 cookie 的域名来提供静态资源是一项标准的性能优化最佳实践。尤其是使用 h1 时，你无法压缩首部，而且有些网站使用的 cookie 大小常常超过单个 TCP 数据包的限度。不过，在 h2 下请求首部使用 HPACK 算法被压缩，会显著减少巨型 cookie（尤其是当它们在先后请求之间保持不变）的字节数。与此同时，禁用 cookie 的域名需要额外的主机名称，这意味着将开启更多的连接。

如果你正在使用禁用 cookie 的域名，以后有机会你可能得考虑消灭它。如果你确实不需要那些域名，最好删掉它们。省一个字节就是一个字节。

6.7.5　生成精灵图

目前，生成精灵图仍是一种避免小资源请求过多的技术（你能看到人们乐意做什么来优化 h1）。为了生成精灵图，开发者把较小的图片拼合成较大的图片，然后用 CSS 选择图片中某个部分展示出来。依据设备及其硬件图形处理能力的不同，精灵图要么非常高效，要么非常低效。如果用 h2，最佳实践就是避免生成精灵图；主要原因在于，多路复用和首部压

缩去掉了大量的请求开销。即便如此，还是有些场景适合使用精灵图。

> **性能优化因人而异**
>
> 如果本节让你感到失望，由于其中的内容，以及前提明确的建议加上用词谨慎的告诫，这也正常。为了最大化 Web 性能，你需要在许多变量之间取舍，包括网络条件、设备处理能力、浏览器能力，**还有**协议限制。这些组成了我们所说的**场景**，大多数开发者的时间远远不够考虑那么多场景。
>
> 怎么办？最佳实践的第一原则就是：**测试**。性能测试与监控是获得最大成果的关键，HTTP/2 也不例外。观察真实用户数据、详尽分析各种条件、查找问题，然后解决它们。要遵循业界推荐的方式，但也不要陷入过早优化的陷阱。应当让数据为你的优化尝试指引方向。

6.7.6 资源预取

资源预取也是一项 Web 性能优化手段，它“提示”浏览器，只要有可能就继续下载可缓存资源，并把这些资源缓存起来。尽管如此，如果浏览器很忙，或者资源下载花的时间太长，预取请求将会被忽略。资源预取可以在 HTML 中插入 `link` 标签实现：

```
<link rel="prefetch" href="/important.css">
```

也可以使用 HTTP 响应中的 Link 首部：

```
Link: </important.css>; rel=prefetch
```

资源预取与 h2 引入的服务端推送并没多少关联。服务端推送用于让资源更快到达浏览器，而资源预取相比推送的优点之一是，如果资源已经在缓存里，浏览器就不会浪费时间和带宽重复请求它。所以，请把它看作 h2 推送的补充工具，而不是将被替代的特性。

6.8 现实情况中的性能

理论与测试当然很不错，但是 IETF 有句谚语说：“代码自己会说话，但是数据的嗓门更大。”下面来看几个现实中使用 HTTP/2 的网站及其最终表现。

6.8.1 性能测量方法论

我们的性能测试方法是，借助 WPT（参见 8.2 节），以下列几个维度的组合来测试每个网站。

测试地点

　分散在各处的测试地点（美国东海岸、美国西海岸以及欧洲）。

浏览器

Chrome 与 Firefox（选择它们是因为可以很容易地禁用 h2，允许 A/B 测试）。

连接方式

模拟的网络连接（有线连接和信号强的 3G 网络）。

测试次数

每个测试运行 9 次（为了得到更好的平均值）。

也就是说，我们对每个网站做 108 次 WPT 测试（3×2×2×9）。这个办法远非完美，也不能百分之百令人信服，但是它能给有兴趣自己做基础性能测试的人提供指导，也不用投入太多时间和资源。

本节大量使用了 WebPagetest 的结果，因此我们需要能够访问到那些结果，以便有效利用它们。如果想在家动手，请点击文中的链接，然后你就可以乐在其中了。

6.8.2　案例1：www.facebook.com

Facebook 的工程师通过对比 h2 与 h1 的测试结果来持续进行改进。显然，他们已经花了很多时间来调优 h2 的服务。例如，当客户端通过 h2 请求他们的主页时，服务器会根据配置撤销域名拆分；这样可以最大程度利用单一连接上的多路复用，以及 h2 的请求优先级管理。

就可感知的性能而言，Facebook 的首页在 h2 下与 h1 相比，开始显示时间快了 33%，提速高达 1.5 秒（见图 6-12）。访问 http://goo.gl/m8GPYO 可以看到这一测试。

如果深入观察图 6-12 中的瀑布流就会看到，这种区别的主要原因是 h2 采用单个 TCP 连接——在 h1 瀑布流中，明显可以看到开启 6 个连接造成的性能损耗。请对比 h1（http://goo.gl/sWjL3M）和 h2 的瀑布流（http://goo.gl/w4vSLg）。

如果仔细看 h2 测试的瀑布流，你会看到大部分渲染所需的关键资源从同一域名加载，也就是传输主体 HTML（www.facebook.com）的那个域名。在 h1 版本中，HTML 从 www.facebook.com 加载，但是大部分 CSS 和 JavaScript 从 static.xx.fbcdn.net 加载。虽然在这种场景下，性能损耗是由 h1 开启新的 TCP 连接造成的，如果点击所包含资源的 h2 瀑布流，就会看到 WPT 显示了依赖项和优先级，如表 6-4 所示。（注意，我们用 Chrome 作为 WPT 代理。这一点很重要，因为在本书写作之际，WPT 显示的流信息因所选择的浏览器会略有不同。）

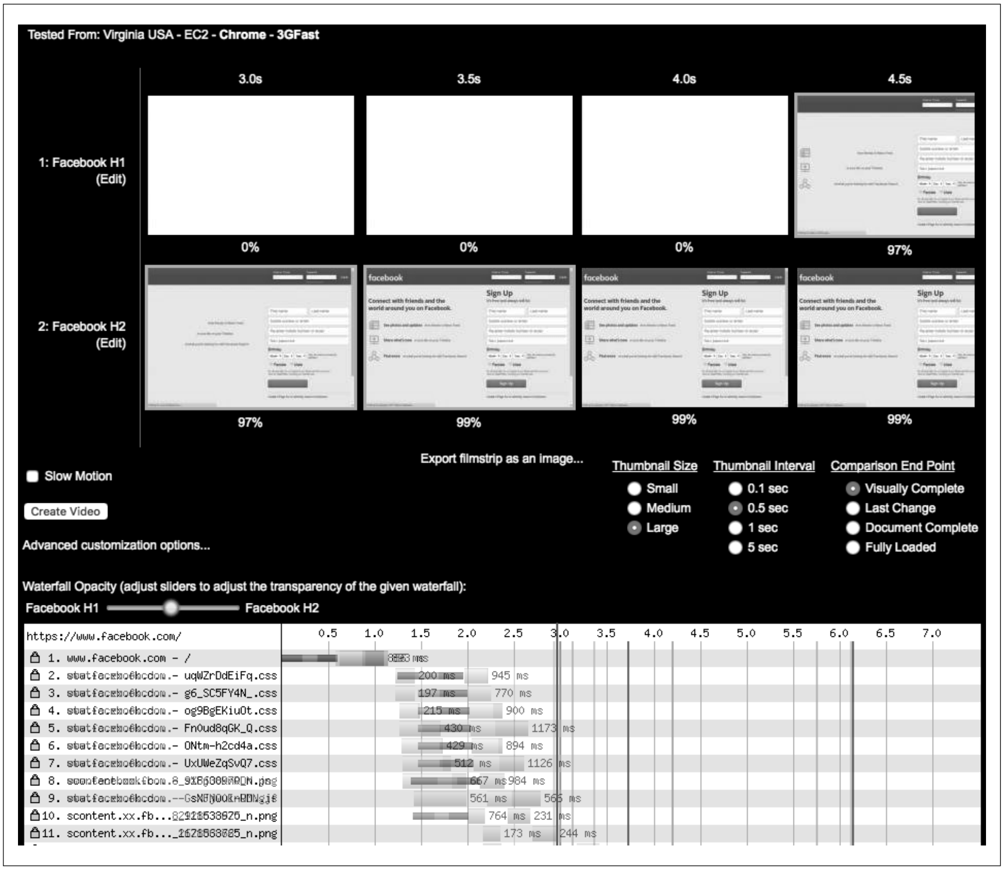


图 6-12：在模拟 3G 网络条件下通过 h1 与 h2 访问 www.facebook.com

表6-4：HTTP/2依赖项与优先级（权重）

URL	优先级	HTTP/2流信息（依赖项）
https://www.facebook.com/	非常高	1，权重 256 依赖于 0，EXCLUSIVE
https://www.facebook.com/rsrc.php/v3/yE/r/uqWZrDdEiFq.css	高	3，权重 220 依赖于 0，EXCLUSIVE
https://www.facebook.com/rsrc.php/v3/yQ/r/g6_SC5FY4N_.css	高	5，权重 220 依赖于 3，EXCLUSIVE
https://www.facebook.com/rsrc.php/v3/yD/r/og9BgEKiuOt.css	高	7，权重 220 依赖于 5，EXCLUSIVE
https://www.facebook.com/rsrc.php/v3/yn/r/Fn0ud8qGK_Q.css	高	9，权重 220 依赖于 7，EXCLUSIVE

（续）

URL	优先级	HTTP/2流信息（依赖项）
https://www.facebook.com/rsrc.php/v3/yE/r/ONtm-h2cd4a.css	高	11，权重 220 依赖于 9，EXCLUSIVE
https://www.facebook.com/rsrc.php/v3/yG/r/UxUWeZqSvQ7.css	高	13，权重 220 依赖于 11，EXCLUSIVE
https://www.facebook.com/rsrc.php/v3/yh/r/sXFjO0knRDN.js	中	15，权重 183 依赖于 13，EXCLUSIVE
https://www.facebook.com/rsrc.php/v3/yb/r/GsNJNwuI-UM.gif	非常低	17，权重 110 依赖于 15，EXCLUSIVE
https://scontent.xx.fbcdn.net/t39.2365-6/851565_602269956474188_918638970_n.png	非常低	1，权重 110 依赖于 0，EXCLUSIVE
https://scontent.xx.fbcdn.net/t39.2365-6/851585_216271631855613_2121533625_n.png	非常低	3，权重 110 依赖于 0，EXCLUSIVE
https://scontent.xx.fbcdn.net/t39.2365-6/851558_160351450817973_1678868765_n.png	非常低	5，权重 110 依赖于 3，EXCLUSIVE
https://www.facebook.com/rsrc.php/v2/ye/r/0qx-vnsuxPL.png	非常低	19，权重 110 依赖于 0，EXCLUSIVE
……		

6.8.3　案例2：www.yahoo.com

www.yahoo.com 在使用 HTTP/2 后效果看来也不错。yahoo.com 的 h2 页面 4 秒就开始显示，然而 h1 版本需要 5.5 秒（见图 6-13）。访问 http://goo.gl/eRUilp 可以看到这一测试。

与 Facebook 的例子相似，它会根据不同客户端决定使用 h2 还是 h1 来传输页面；另外，www.yahoo.com 加载资源时略有不同。使用 HTTP/2 时，传输 HTML 及所需资源使用的都是 www.yahoo.com；但是使用 h1 时，HTML 使用域名 www.yahoo.com 传输，页面所需静态资源却使用域名 s.yimg.com 传输。如果使用 h1，浏览器针对域名 s.yimg.com 开启 6 个链接，这也会造成相当长的延迟。另外，使用 h2 时页面元素会以不同次序加载，这也是 DOM 完成时间与 PLT 变得更短的原因，如图 6-13 所示。Yahoo! 网站从位于美国的 WPT 代理加载时，性能有明显提升。有趣的是，如果使用爱尔兰的 WPT 代理，通过 h1 加载反而更快。此时，h2 页面要 7.5 秒才开始显示，h1 却只要 4.5 秒。详情参见 https://goo.gl/GrySYa。

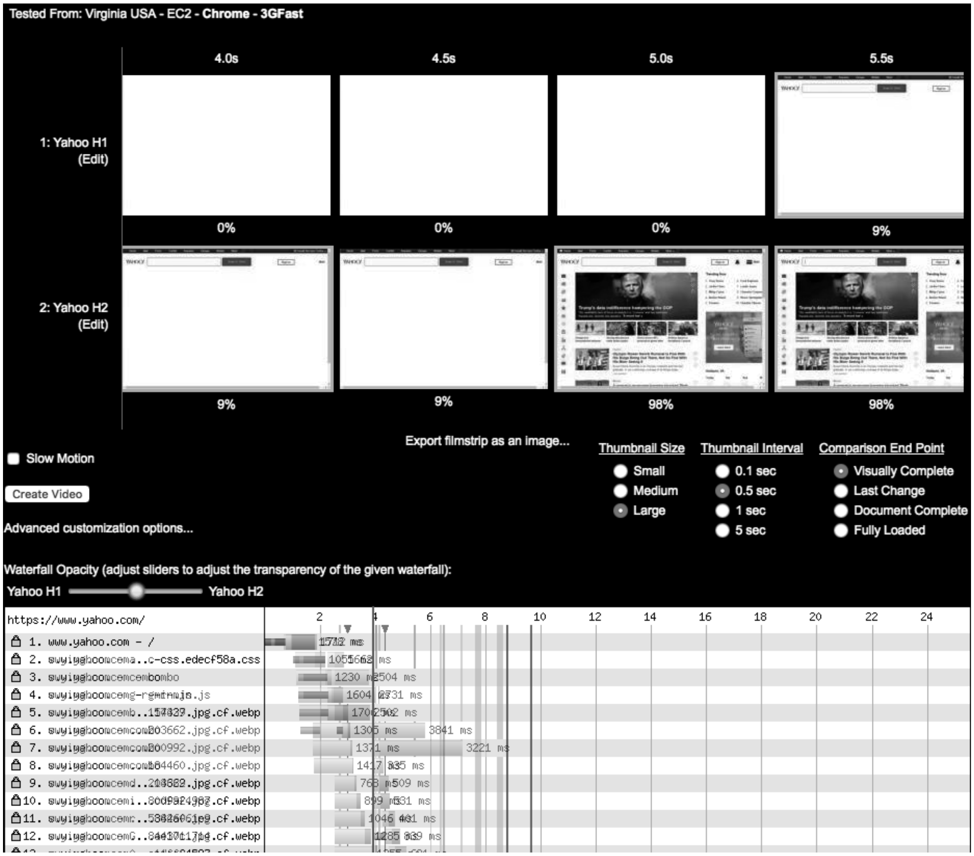


图 6-13：在美国模拟 3G 网络条件下通过 h1 和 h2 访问 www.yahoo.com

从图 6-14 可以看出，貌似 Yahoo! 生成爱尔兰版主页的方式相比美国版稍有不同。在这种情况下，通过 h2 加载页面时，仍然保留了域名拆分。大部分引用的资源（例如 https://s.yimg.com/zz/combo?/os/stencil/3.1.0/styles-ltr.css&/os/fp/atomic-css.edecf58a.css）通过其他的域名访问，而不是使用与主体 HTML 相同的域名。这些数据说明，部分测试用例使用 h2 时明显更慢，即使一般情况也是会有例外的。

但是，为什么 h2 居然会更慢？你可能会摆手否定它，并借用一些统计学概念说样本太小没有意义之类。但是，这也许会掩盖重要事实：h2 并不是在任何场景下都更快。在某些特殊场景下，使用多个连接（而不是单个）以及具体的网络环境，可能是 h1 反而更快的原因；网络丢包是 h2 的命门，一次丢包机会就会让它的所有优化泡汤。其他情况下，问题往往来源于网站自身的结构。在 h2 的早期阶段，它也可能源于尚未实现的 bug。不管怎么说，在特定场景下看到性能下降其实是符合预期的，况且这种情况几乎总是可以定位的。

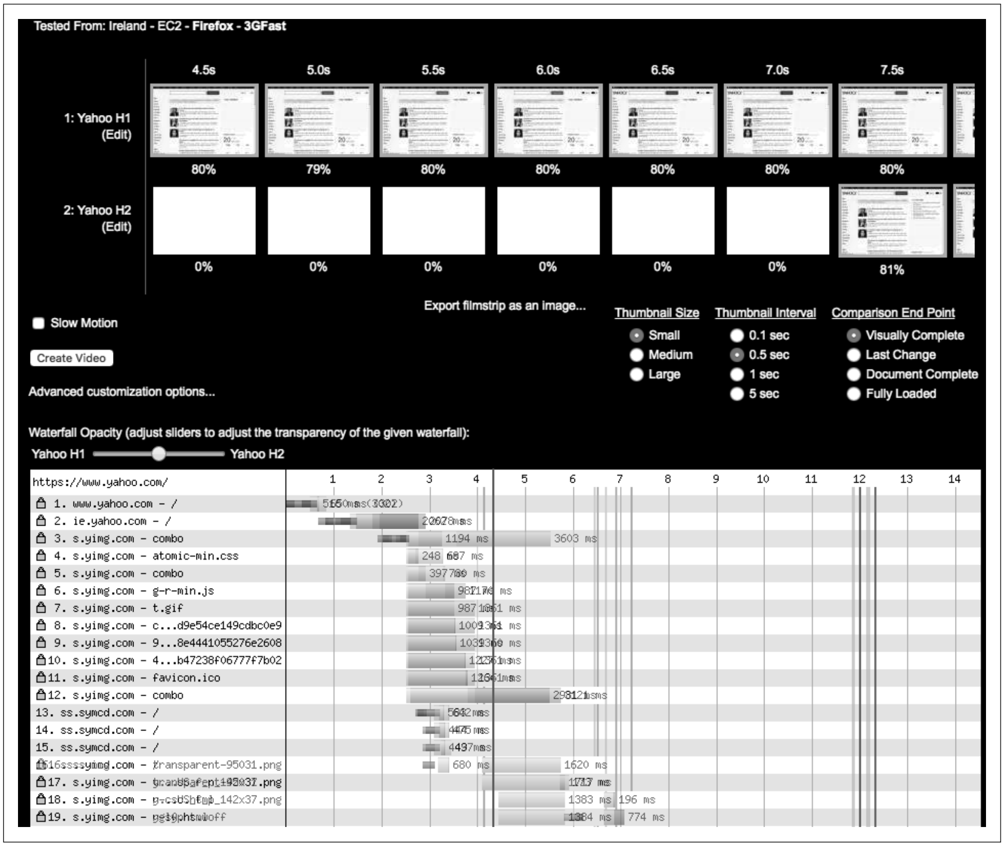


图 6-14：在爱尔兰模拟 3G 网络条件下通过 h1 和 h2 访问 www.yahoo.com

6.9 小结

你选择看这本书，很可能是因为性能对你很重要。HTTP/2 提供了我们所需要的性能，但同样重要的是理解它为什么能提升性能，以及有时反而变得更慢的原因。跟其他任何性能调优尝试一样，h2 可能也要经历编码、测试、分析以及优化的迭代过程。即便你的网站开启了 h2，也确实运作得比之前更好，但如果再稍微深入分析一下，你或许会发现更多可以优化的地方。

第 7 章
HTTP/2 实现

HTTP/2 获得成功的原因之一就是主流工具对它提供了各种实现支持。甚至在 RFC 定稿之前，就有相当多的互联网用户在暗地里使用 h2，默默享受“推动 Web 进化的法宝”带来的福利了。这种支持并不局限于市场份额小的实验性浏览器和 Web 服务器；相反，最流行的浏览器、服务器、代理、缓存以及命令行工具早已包含了 h2。无论你是定位自己的 Web 用户，还是选择合适的 Web 服务器或代理，如果要制订开发计划，一定要清楚了解哪些实现可用，以及各种实现提供了哪些功能。

由于互联网发展速度远快于本书的写作进度，我们不可能触及每一项功能，也无法穷举所有支持的软件。但是通过阅读本章，你至少可以对各种特性有个大体上的认识，并能找到好的着手点。

7.1 桌面Web浏览器

让大家**无需**学习任何预备知识即可直接上手使用 h2，这就是浏览器的开发目的。只要使用现代流行的 Web 浏览器，那么你就已经在日常生活中使用 h2 浏览网页了。启用 HTTP/2 之后，Web 的工作原理与以往只有很小的区别。尽管如此，了解这些区别以及与此相关的不同特性，对你的开发和调试工作是非常重要的。

7.1.1 只支持TLS版

本章提到的所有浏览器在进行 HTTP/2 传输时都需要使用 TLS（HTTPS），即使事实上 HTTP/2 规范本身并没有强制要求 TLS。请参阅 4.2 节的附注栏“TLS 是必要的吗”对这一话题的讨论。

7.1.2 禁用HTTP/2

HTTP/2 毕竟是新鲜事物，在碰到问题时，你可能想要在自己的网站上禁用它。或者，你可能想要对比 h2 开启和关闭的不同请求轨迹。不管是什么情况，你都需要有办法在浏览器里启用 / 禁用 HTTP/2。不幸的是，并非所有浏览器都允许你这么做。表 7-1 列出了支持启用 / 禁用 HTTP/2 的浏览器。

表7-1：支持启用/禁用HTTP/2的浏览器

浏览器	第一个支持HTTP/2的版本	支持启用/禁用	支持服务端推送	支持连接归并	支持协议调试	支持获取beta版本
Chrome	41（2015 年 3 月）	是	是	是	是	是
Firefox	36（2015 年 2 月）	是	是	是	是	是
微软 Edge	12（2015 年 7 月）	否	是	否	是	是
Safari	9（2015 年 9 月）	否	是	否	否	是
Opera	28（2015 年 3 月）	是	是	是	是	是

7.1.3 支持HTTP/2服务端推送

服务端推送是 h2 中最令人兴奋也最难正确使用的特性之一。因为普通的页面传输并不需要它，所以最初的 h2 实现版本有时不支持它。然而，所有的主流浏览器（见表 7-1）都已经支持了此特性。

7.1.4 连接归并

如果需要建立一个新连接，而浏览器支持连接归并，那么通过复用之前已经存在的连接，就能够提升请求性能。这意味着可以跳过 TCP 和 TLS 的握手过程，改善首次请求新域名的性能。如果浏览器支持连接归并，它会在开启新连接之前先检查是否已经建立了到“相同目的地”的连接。在这种场景下，**相同目的地**具体指的是：已经存在连接，其证书对新域名有效，此域名可以被解析成那个连接对应的 IP 地址。如果上述条件都满足，那么浏览器会在已建立的连接上向该域名发起 HTTP/2 请求。

7.1.5 HTTP/2调试工具

在使用 h2 时，能看到它后台的信息有时候也很重要。一些浏览器在自己的工具集里对 h2 专门提供了很棒的支持。关于这一主题，更多信息可参见第 8 章。

7.1.6 浏览器beta版本

其实这并不局限于 HTTP/2 的特性，如果能获取浏览器的 beta（或更早期的）版本，我们就能够预知变化，并且可以体验协议的最新进展。

如果想全面了解支持 h2 的浏览器的最新列表，参见 caniuse.com。[1]

7.2 移动端

如今，移动端浏览器一般会尽量与其桌面版保持步调一致（见表 7-2）。

表7-2：移动端浏览器的支持情况

浏览器	操作系统	最早支持HTTP/2的版本	支持服务端推送	支持连接归并
Chrome	iOS	41（2015 年 3 月）	是	否
Safari	iOS	9.2（2015 年 9 月）	是	否
Chrome	Android	12（2015 年 7 月）	是	是
Android	Android	53（2016 年 10 月 ）	是	是
微软 Edge	Windows Mobile	12（2015 年 7 月）	是	否

截至目前，还没有移动端浏览器支持禁用 h2。

7.3 移动端应用支持

自从 2015 年 6 月 XCode 更新[2]之后，苹果公司在 iOS 中提供了对 h2 的支持；默认情况下，在使用 NSURLSession 时，网络传输安全模块就会允许原生 iOS 应用充分利用 h2 协议。对于安卓应用，它们需要使用兼容 h2 的第三方库，比如 OkHttp[3]，并且必须通过 TLS 连接到支持 h2 的 Web 服务器上。

7.4 服务器、代理以及缓存

如果要通过 h2 传输内容，我们有几个选择。支持 HTTP/2 的网络设施大致有以下两类。

Web服务器

通常所说的提供静态和动态内容服务的程序。

代理/缓存

一般处在服务器和最终用户之间，可以提供缓存以减轻服务器负载，或进行额外加工，或两者皆有之。许多代理也能扮演 Web 服务器的角色。

注 1：http://caniuse.com/#search=http2

注 2：https://lukasa.co.uk/2015/06/HTTP2_Picks_Up_Steam_iOS9/

注 3：http://square.github.io/okhttp/

为了创建高性能、高可用的网站，这些设备通常会组合起来使用。对于比较小的网站来说，代理设备可能不是必需项。

在选择 HTTP/2 服务器时，我们需要检查、评估一些关键点。除了基本的通用性能、操作系统支持、学习曲线、可扩展性以及稳定性，还应当关注 Web 请求的**依赖项**和**优先级**，以及对**服务端推送**的支持。

服务端推送通常有两种实现方式：静态文件包含，或者使用 Link 首部以及请求标签。这两种方式都不完美，但是如果配置得当，它们可以提升性能。完美的服务端推送需要与浏览器协同，以便知道哪些资源已经在本地缓存里，由此避免不必要的推送。然而到目前为止，还没有服务器支持这种协同操作。

使用 Link 首部

服务器告诉代理哪些资源需要推送时，一种办法是对于需要推送的每个资源发送一个 Link 首部。虽然实现细节会因服务器和代理类型的不同而有区别，但是通常会在响应中添加如下首部：

```
Link: </script.js>; rel=preload
```

它的意思是 script.js 会被推送。

为了最大程度利用 HTTP/2 以提升 Web 性能，对 Web 请求的依赖项和优先级的有效管理是重要的工具之一。这也是设计支持 h2 的服务器时最有挑战也最值得花心思的部分。如果想大体上了解哪种服务器最适合你的使用场景，也许可以测试它们在几款不同浏览器上的表现。

表 7-3 列举了若干常见的服务器，以及它们在本书写作之际的兼容性。

表7-3：兼容HTTP/2的网络端点

服务器	类型	支持服务端推送
Apache	服务器	是
Nginx	服务器 / 代理 / 缓存	否
IIS	服务器	是
Jetty	服务器	是
h2o	服务器 / 代理	是
Squid	代理 / 缓存	是
Caddy	服务器	否
Varnish	代理 / 缓存	否
Traffic 服务器	代理 / 缓存	是

7.5　内容分发网络

内容分发网络（CDN）是反向代理服务器的全球性分布式网络，它部署在多个数据中心。CDN 的目标是通过缩短与最终用户的距离来减少请求往返次数，以此为最终用户提供高可用、高性能的内容服务。它是互联网规模不断扩张、性能不断提升的关键一环。CDN 服务的范围很广，不仅有针对小流量 Web 站点的“免费使用”服务，也有为全球规模最大的 Web 站点提供高性能、高可靠性以及高安全性内容的企业级服务。

大多数主流 CDN 是支持 HTTP/2 的，虽然协议支持的完整程度以及功能特性会因厂商不同而有所差异。与评估 Web 服务器类似，选择 CDN 时要考虑的两大要点是：对服务端推送的支持，以及它们处理优先级的方式。这两点对现实世界中的 Web 性能影响重大。

7.6　小结

鉴于 h2 还如此年轻，它如今得到广泛的支持真是了不起。目前最流行的 Web 服务器、代理与 CDN，以及超过 70% 的正在使用的浏览器，都已经完全支持 h2 了。但是另一方面，服务器端推送之类的 h2 特性仍处于初期阶段，对请求依赖项和优先级的优化仍在继续。毫无疑问，我们不久就会看到 h2 在这些领域的进展。

第 8 章

HTTP/2调试

HTTP/1 协议已经有近 20 年历史了，对应的监控和调试的工具也发展了 20 年。如今 HTTP/2 来了，老办法也已经过时了。如果没有工具更新或者替代品出现，这将是你调试 h2 的重大阻碍。不过，h2 这头怪兽尽管看起来和 h1 差不多，二者毕竟还是有区别的。之前能用的东西现在大多也还能用，但是区别也很大。好在无论是 HTTP/2 的专用工具，还是正在使用的工具集，都涌现出众多新的测试和调试工具。

8.1　浏览器开发者工具

大多数现代的浏览器都包含 Web 开发者工具，它们对调试 h2 非常有用。随后几节给了一些例子，说明如何用 Chrome 和 Firefox 的开发者工具来跟踪 h2 的流、确定 h2 中的优先级和依赖，以及理解服务端推送。请注意，相似的概念也可以应用于 Safari、Edge 或者 Opera 之类流行的浏览器。

8.1.1　Chrome开发者工具

在过去几年中，Chrome 浏览器已经获得开发人员的认可，部分原因是 Web 开发者工具的功能在逐步增加。[1] 熟练掌握开发者工具，可以为将要遇到的技术难题提供解决思路。从 Chrome 菜单里的视图（View）→开发者（Developer）→开发者工具（Developer Tools）启用开发者工具。

注 1：https://developers.google.com/web/tools/chrome-devtools/network-performance/resource-loading

1. net-internals

你可以通过在地址栏里输入 `chrome://net-internals` 访问 Chrome 的 net-internals[2] 功能。这些工具可以查看网络数据，包括捕获 / 导出 / 导入底层网络数据、检查网络和 DNS 日志、以图形界面显示网络活动。

net-internals 工具集可以用来捕获 HTTP/2 流量，有助于解释下列 HTTP/2 概念：

- 流 ID
- 优先级
- 依赖
- 服务端推送的承诺（PUSH_PROMISE）
- 会话 `PING`

以下是捕获 HTTP/2 流量的步骤。

(1) 在 Chrome 的地址栏输入 **`chrome://net-internals`**。
(2) 在边栏中选择 `HTTP/2`。
(3) 打开一个新标签页，在地址栏输入你喜欢的 URL。
(4) 返回到 net-internals 标签页，你将看到一张列表，上面是与所有启用 HTTP/2 协议的主机名的活跃会话。
(5) 点击所访问 URL 主机名右侧的 `ID` 列中对应的链接（见图 8-1）。

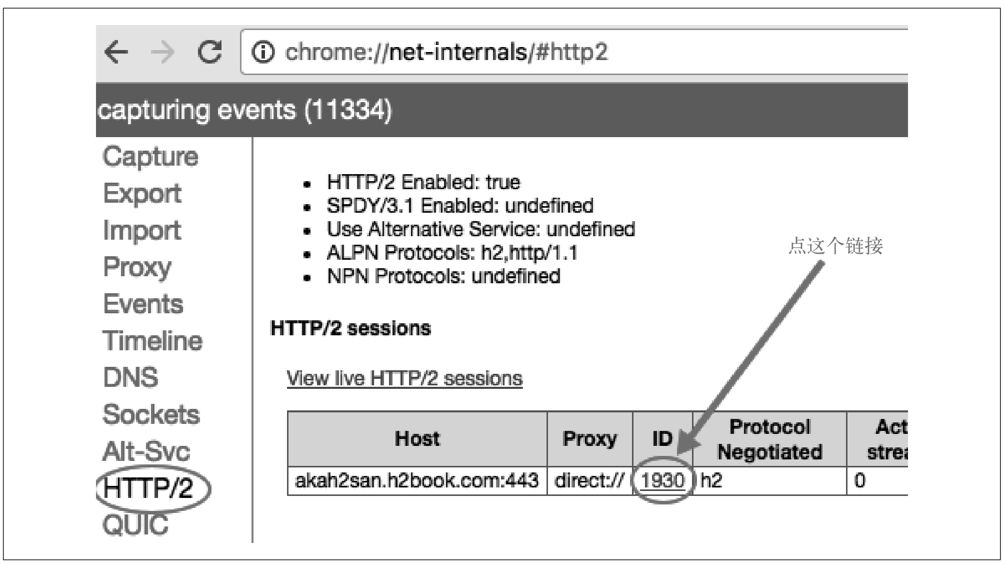


图 8-1：net-internals `HTTP/2`

注 2：https://www.chromium.org/developers/design-documents/network-stack/netlog

(6) 这时页面内容将会切换到 `Events`；勾选主机名左边的复选框（见图 8-2）。

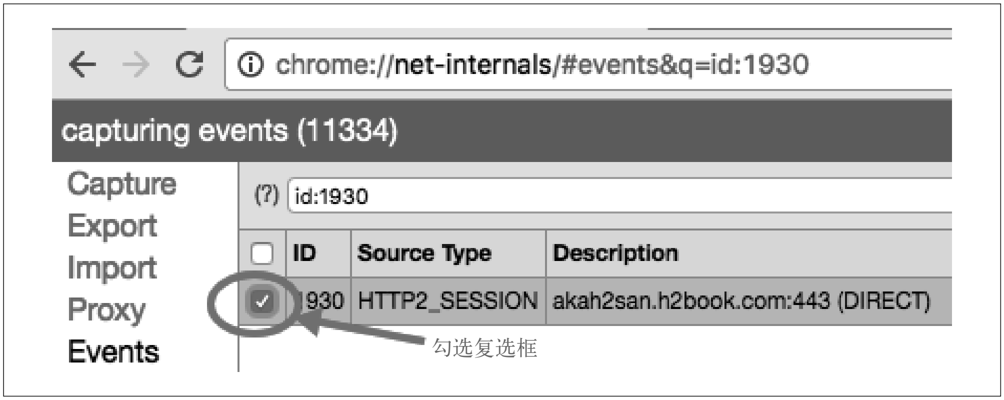


图 8-2：net-internals `Events`

(7) 在页面右边，你将看到 Web 服务器和客户端之间所有捕获到的网络请求。

来看看捕获到的字段。每个事件（event）以此开始：

```
t=timestamp [st=    milliseconds] EVENT_TYPE
```

例如：

```
t=123808 [st=    1]   HTTP2_SESSION_SEND_HEADERS
```

- 第一个字段 `t` 代表时间戳，单位为毫秒，从浏览器会话开始计数。例如，123808 代表 HTTP/2 会话在浏览器会话开始之后运行了 123.8 秒。
- 第二个字段 `st` 代表从 HTTP/2 会话开始的相对时间。例如，1 代表这个事件在 HTTP/2 开始之后 1 毫秒内发生。
- 第三个字段代表所记录事件的类型。

> 有些 Chrome 的 h2 事件类型可以直接对应到 h2 的帧类型。例如，`HTTP2_SESSION_RECV_PUSH_PROMISE` 对应 PUSH_PROMISE 帧，`HTTP2_SESSION_PING` 对应 PING 帧，等等。

下面来看一个浏览器捕获到的示例，学习如何检阅捕获到的日志，并能从中获取哪些 HTTP/2 信息：

```
t=791301 [st=    1]    HTTP2_SESSION_SEND_HEADERS ❶
                       --> exclusive = true
                       --> fin = true             ❷
                       --> has_priority = true    ❸
                       --> :method: GET           ❹
```

```
                    :authority: akah2san.h2book.com
                    :scheme: https
                    :path: /
                    cache-control: max-age=0
                    upgrade-insecure-requests: 1
                    user-agent: Mozilla/5.0 (Macintosh; Intel Mac..
                    accept: text/html,application/xhtml+xml,...
                    accept-encoding: gzip, deflate, sdch, br
                    accept-language: en-US,en;q=0.8
                    cookie: [30 bytes were stripped]
                    if-none-match: "11168351bd3324ad3e43ed68195063c5:1464989325"
                --> parent_stream_id = 0  ❺
                --> stream_id = 1         ❻
                --> weight = 256          ❼
...
```

❶ 这一行是事件信息行，如之前所述。

❷ `fin = true` 表示接下来没有更多的 head 帧了。

❸ 这个请求设置了优先级。

❹ 帧的 HTTP 首部从这儿开始。

❺ 对应的父级流的 ID 是 0。

❻ 当前流的 ID 是 1（客户端发起的第一个请求）。

❼ 依赖的相对权重是 256。

通过查看在 net-internals 中的事件，你可以对网络上发生的一切了如指掌，还可以观察到协议内部的实现。

> 如果你觉得很难读懂这份 Chrome 输出的日志，并不是只有你是这样——有一些工具可以帮助你。Rebecca Murphy 创建了一个非常有用的小工具叫 chrome-http2-log-parser[3]，正如所介绍的那样，这个工具把在 net-internals 中捕获的 HTTP/2 日志变成“真正有用的东西”。推荐你使用它。

2. 服务端推送的可视化

Chrome 开发者工具中的 Network 栏，有助于简单直观地跟踪客户端和服务端的通讯，它按下面表格的形式展示了若干信息：

- 资源名
- 资源大小
- 状态码
- 优先级
- 总加载时间

注 3：https://github.com/rmurphey/chrome-http2-log-parser

- 使用时间线方式分解加载时间

来看个例子。加载网页 https://akah2san.h2book.com/（一个使用 HTTP/2 和服务端推送的简单网页）。

在 Network 标签页中，你看到的如图 8-3 所示。

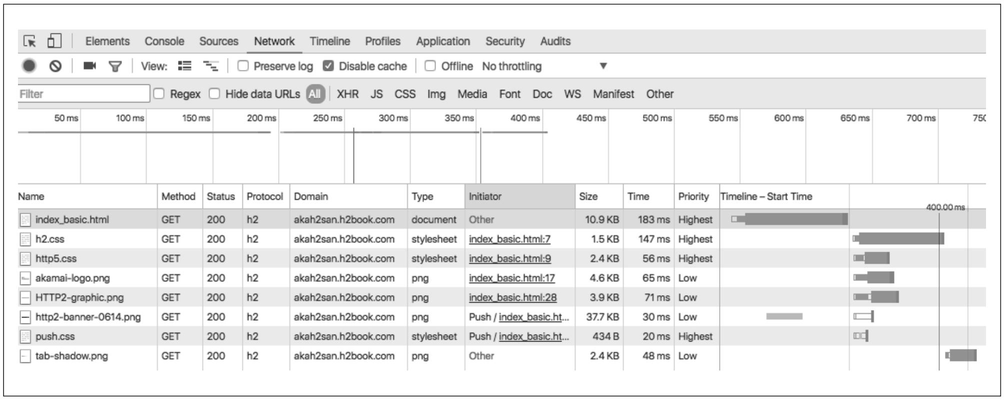


图 8-3：服务端推送时间线

Network 栏页显示，当前 HTML 加载了 3 个样式文件和 4 个 PNG 图片。这 7 个资源中，有 2 个（/resources/push.css 和 /resources/http2-banner-0614.png）是被“推送”到浏览器的，其余 5 个按普通方式加载。

如果把鼠标悬停在面板右侧瀑布流中的资源上，就会看到资源完整加载过程中各个阶段的详细时间。下面解释了悬浮框里展示的信息（见图 8-4）：

- Connection Setup（连接设置）
  - Queueing——请求被渲染引擎或者网络层延迟的时间
  - Stalled——请求可以被发送出去之前等待的时间
- Request/Response（请求 / 响应）
  - Request Sent——发送请求包含的数据花费的时间
  - Waiting (TTFB)——等待初始的响应花费的时间，也就是所说的 TTFB（首字节时间）；这个数字包括等待服务器传输响应的时间，以及往返服务器的延迟
  - Content Download——接收响应的数据所花费的时间
- 总时间（标签上显示为 Explanation）

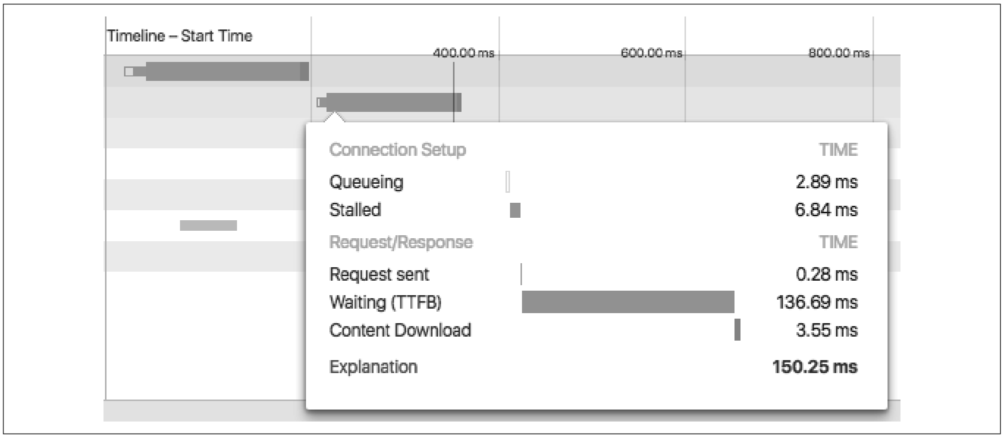


图 8-4：服务端推送时间线 h2.css

把鼠标悬停在服务端推送的资源上面，就会看到图 8-5 所示的信息：

- Server Push（服务端推送）
  - Receiving Push——接收服务端推送资源的全部字节所花的时间
- Connection Setup（连接设置）
  - Queueing——请求因渲染引擎或者网络层而延迟的时间
- Request/Response（请求 / 响应）
  - Reading Push——浏览器从临时缓存中读取之前服务端推送资源所花的时间
- 总时间（标签上显示为 Explanation）

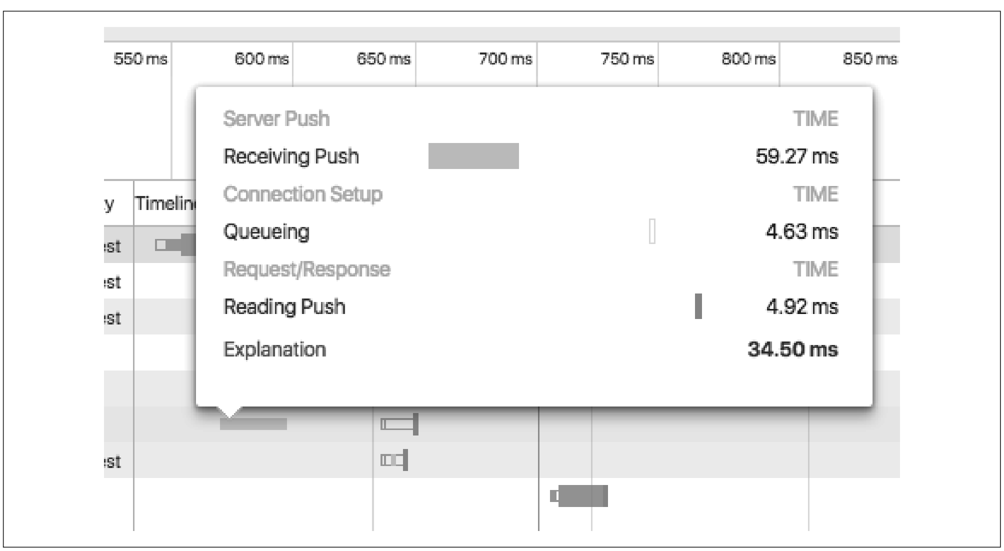


图 8-5：服务端推送时间线 push.css

3. Chrome会话密钥日志（session key logging）

Chrome 和 Firefox 都提供了记录 TLS 会话密钥的功能，会话密钥是为指定的连接进行加密需要用到的。如果使用像 Wireshark（参见 8.7 节）这种外部工具检查 HTTP/2 流量和观察 HTTP/2 帧，会话密钥是很有用的。要开启这个功能，你需要在启动浏览器之前，将会话密钥记录日志文件的路径设置到环境变量里。例如，在 OS X 上可以这么做：

```
$ SSLKEYLOGFILE=~/keylog.txt
$ open /Applications/Google\ Chrome.app/Contents/MacOS/Google\ Chrome
```

Mozilla 为这个设置提供了很棒的介绍：https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/Key_Log_Format。

8.1.2 Firefox开发者工具

Firefox 浏览器提供了各种各样的工具来帮助 Web 开发者。后面几节将介绍 Firefox 的一些功能，它们是调试通过 HTTP/2 传输的 Web 内容的利器。

1. 记录HTTP会话日志

记录 HTTP 会话日志对调试的帮助很大。Firefox HTTP 会话日志记录功能可以帮你深入了解数据传输中发生的故事。现在浏览器还没有**内置**直接获取这些记录的功能，所以需要在命令行里做些操作。

下面用 Firefox 的 HTTP 会话记录功能来捕获一些 HTTP/2 的流量，它清楚地展现了一些 HTTP/2 概念，如：

- 流 ID
- 优先级
- 依赖
- 服务端推送的承诺（PUSH_PROMISE）
- 会话 `PING`

如果要打开日志记录功能，可以在 Windows 的命令行窗口运行如下命令：

```
cd c:\
set NSPR_LOG_MODULES=timestamp,nsHttp:5,nsSocketTransport:5,nsStreamPump:5, ^
  nsHostResolver:5
set NSPR_LOG_FILE=%TEMP%\firefox_http_log.txt
cd "Program Files (x86)\Mozilla Firefox"
.\firefox.exe
```

或者在 mac OS 终端里面这么做：

```
export NSPR_LOG_MODULES=timestamp,nsHttp:5,nsSocketTransport:5, \
  nsStreamPump:5,nsHostResolver:5
export NSPR_LOG_FILE=~/Desktop/firefox_http_log.txt
```

```
cd /Applications/Firefox.app/Contents/MacOS
./firefox-bin
```

一个名为 firefox_http_log.txt 的文本文件将保存在 `NSPR_LOG_FILE` 这个环境变量所指的路径。如果你设置了 `GECKO_SEPARATE_NSPR_LOGS=1`，每个子进程会生成自己的日志文件。每个日志文件都会使用你在 `NSPR_LOG_FILE` 这个环境变量中指定的文件名，但是会带上“.child-*X*”的后缀，*X* 是每个子进程的对应编号。

你可以通过修改 `NSPR_LOG_MODULES` 的值控制日志文件的日志级别。具体有如下两种方式。

- 减小写在模块右边的数字。例如，`nsHttp:3` 表示需要的日志信息比 `nsHttp:5` 更少。可用的调试值参见表 8-1。
- 从列表中去掉一个模块。例如，`NSPR_LOG_MODULES=timestamp,nsHttp:5` 的日志比 `NSPR_LOG_MODULES=timestamp,nsHttp:5,nsSocketTransport:5,ns StreamPump:5` 更少。使用 `NSPR_LOG_MODULES=all:5` 可以获得最多的日志。

Firefox Log Modules（https://mzl.la/2pcgU2o）提供了一张更完整的列表，列出了此处可以使用的模块。

表8-1：`NSPR_LOG_FILE`日志级别

级别	描述
0	不输出日志
1	重要的日志；总要被输出的日志
2	错误
3	警告
4	调试消息；提示
5	所有日志

2. Firefox会话密钥日志

Firefox 像 Chrome 一样，也可以记录 TLS 会话密钥。详情参见 8.1.1 节关于 Chrome 会话密钥日志的内容。

8.1.3 在iOS上使用Charles Proxy调试h2

因为 TLS 加密、iOS 的安全模型，以及 iOS 中一般不提供足够的可见性，所以在 iOS 设备上调试 h2 有些曲折。本节解释如何使用运行 Charles Proxy[4] 的计算机来调试 iOS 设备。顾名思义，Charles Proxy 是一个代理（proxy）。把它设置为设备访问网络的代理，就可以清

注 4：https://www.charlesproxy.com/

楚地看到当前正在发送的请求和接收的响应。这种能力在 Chrome 以及 Firefox 之类的浏览器里已经有了，但是 iOS 上的浏览器（如 Safari）和 iOS 原生应用并不具备。

> 你需要使用 Charles Proxy 4 及以上版本，以确保它可以支持 HTTP/2。另外，尽管 Charles Proxy 值得加到工具箱里，但它不是免费的。为了跟着本书一起动手，你可以使用限期试用版本。nghttp2（参见 8.4 节）的代理模式可以当成免费替代品。

调试过程如下：在计算机上安装 Charles Proxy，将它的根证书安装到要使用代理的设备上（这样 Charles Proxy 可以做“中间人”，解码 TLS 数据），最后在 iOS 设备上设置代理参数，指向运行 Charles Proxy 代理的计算机的 IP 和端口。下面将分别介绍如何在 iOS 模拟器和真机上开展调试。

1. iOS模拟器

要在同一台机器上设置 Charles Proxy，并用 iOS 模拟器来调试 h2，按如下步骤进行。

(1) 退出 iOS 模拟器。

(2) 启动 Charles Proxy。

(3) 打开帮助（Help）菜单，选择 SSL Proxying → Install Charles Root Certificate in iOS Simulators（把根证书安装到本机所有的 iOS 模拟器中）。

启动 iOS 模拟器之后，应该就可以通过采用 TLS 代理模式的 Charles Proxy 访问 TLS 网站了。

2. iOS真机

要在 iOS 设备上调试 HTTPS 数据，你需要在计算机上启动代理，然后更改 iOS 设备的网络设置，才能使用计算机上的代理服务。

> 最好安排一台调试专用的 iOS 设备，因为安装 Charles 的代理根证书可能会覆盖在 iOS 设备上已经有的配置描述文件[5]。

在提供代理服务的计算机上如下操作。

(1) 启动 Charles Proxy。

(2) 在 Proxy 菜单项上选择 Proxy Settings，并将端口号设置为 8888（如果连接到 8888 端口时出了问题，可以使用 Dynamic Port）。

注 5：配置描述文件（configuration profiles），位置在设置→通用→描述文件与设备管理→配置描述文件。
——译者注

(3) 在 Proxy 菜单项上选择 SSL Proxy Settings，点击 Add，在 Host 中输入你想要监听的主机名（还可以使用 * 来表示监听所有主机名）。
(4) 在 Proxy 菜单项上选择 Access Control Settings，并添加 iOS 设备的 IP 地址（或者输入 0.0.0.0/0，表示允许所有设备连接）。
(5) 在 Help 菜单项选择 Local IP Address，输入你本机的 IP 地址，然后关闭窗口（你将使用这个 IP 地址作为 iOS 设备的代理服务器）。
(6) 在 File 菜单上选择 New Session，开始记录流量。

上述过程的更多信息参见 https://www.charlesproxy.com/documentation/proxying/ssl-proxying/。

下一步，在 iOS 设备上操作。

(1) 打开设置→无线局域网（Wifi settings），然后点击你所连接到的 WiFi 网络名称右边的信息图标，这个图标看起来像 (i)。
(2) 在打开的列表里，往下拉动到 HTTP 代理（HTTP PROXY）一节，选择手动（Manual）。
(3) 在显示出来的服务器（Server）输入框里，填入提供代理服务的计算机的 IP，然后在端口（port）栏输入 8888（如果你使用 Dynamic Port 的话，输入具体采用的端口号[6]。
(4) 按下 Home 键，启动 Safari，在地址栏输入 chls.pro/ssl（或者 http://www.charlesproxy.com/getssl/）以安装 Charles 的代理根证书（解码 TLS 流量所需要的）。

现在你应该可以通过启用 SSL 代理的 Charles 工具来访问 TLS 加密的网站了（除了像 Apple 原生应用那样使用内置证书的某些应用）。

8.1.4　在Android上调试h2

你要先做一点设置才能开始在 Android 上调试 h2。在 Android 设置上，打开设置（Settings），找到开发者选项（Developer Options）一节，打开 USB 调试（USB debugging）功能（如果你使用 Android 4.2 或者更高版本，可能会找不到开发者选项，这时你需要启用它[7]）。

做了这些之后，在你的开发机上执行如下步骤。

(1) 打开 Chrome（确认你已经登录了你的账号，因为这种调试方法在隐身模式或者访客模式下不管用）。
(2) 选择视图（View）→开发者（Developer）→开发者工具（Developer Tools），在右边的“...”菜单里选择 More tools → Remote devices；这里可以看到所有连接上的远程设备的状态（确保勾选了 Discover USB devices）。

注 6：即之前配置 Dynamic Port 时 Charles 自动选择的端口号。——译者注
注 7：https://developer.android.com/studio/run/device.html

(3) 用 USB 线把你的 Android 设备连接到开发机的 USB 接口上（不要使用任何的 USB-hub）。
(4) 第一次将 Android 设备连接到计算机上的时候，需要授权连接；在 Android 设备上弹出的“允许 USB 调试”（Allow USB debugging）对话框上点击确认按钮，确认授权；在授权设备之后，你将看到此 Android 设备处于连接（Connected）状态。

现在你可以详查该设备上的流量了。你还可以勾选 toggle screencast 按钮，在开发者工具（DevTools）面板中查看 Android 设备屏幕上显示的内容。

8.2 WebPagetest

WebPagetest 是一款免费的基于 Web 的性能监测工具，用来全方位测量网站的性能。它借助分布在世界各地的服务器集群上运行的 Web 浏览器，测试网站在不同的网络环境下和不同类型的浏览器上的表现如何。其他值得关注的功能还有：

- 多种脚本化的方式执行测试，来模拟一个完整的浏览器会话；
- 保存 Web 页面加载过程的截图和视频，供之后的各次测试中比较；
- 获得完整的跟踪信息，用于 Wireshark 之类的工具进行分析（参见 8.7 节）；
- 支持不同的网络参数，以限制带宽、提高延迟，以及添加丢包率。

要检验你的变更在各种不同场景下的表现如何，这是一门利器。这个话题可以更详细地讨论，此处暂不涉及。想要进一步了解的话，建议阅读 Rick Viscomi、Andy Davies 和 Marcel Duran 的著作《WebPagetest 应用指南》。

8.3 OpenSSL

OpenSSL[8] 是 SSL 和 TLS 协议的一种开源实现，它可以作为软件类库（通过 Apache 和 BSD 许可协议发行），供应用程序使用来保护通信安全。本节主要关注它的命令行工具，也就是 `openssl`，以及如何使用它来调试 HTTP/2。

OpenSSL命令

由于很多 Web 浏览器都只支持 HTTPs 加密的 HTTP/2，在检验 Web 服务器的 SSL 证书是否满足 HTTP/2 要求的时候，`openssl` 命令非常有用。验证方式如下（只需要将 akah2san.h2book.com 替换成你想验证的域名）：

```
$ echo | openssl s_client -connect \
        akah2san.h2book.com:443 -servername akah2san.h2book.com \
          -alpn spdy/2,h2,h2-14 | grep ALPN
...
ALPN protocol: h2
```

注 8：https://www.openssl.org/

最后的 `| grep ALPN` 会过滤输出，只剩下几行。如果你省略了这个命令，你将看到 `openssl s_client` 这个命令的所有输出，其中包含调试 TLS 配置的相关信息。完整的输出里面包含证书链、证书、协商使用的加密协议，还有其他各种细节。花些时间研究这个工具，你会大有收获。

8.4 nghttp2

nghttp2（https://nghttp2.org/）是 HTTP/2 和它的首部压缩算法 HPACK 的一个 C 语言实现。

HTTP/2 的分帧层以可重用的 C 类库的方式实现。基于此，nghttp2 提供了一整套工具，如表 8-2 所示。

表8-2：nghttp2工具集

工具	描述
nghttp	命令行客户端
nghttpd	服务器
nghttpx	代理
h2load	负载测试工具
inflatehd	HPACK 命令行首部解压工具
deflatehd	HPACK 命令行首部压缩工具

本节主要关注命令行的客户端，即 `nghttp`。

使用nghttp

你可以使用 `nghttp` 请求一个 HTTP/2 的 URL 进行调试，并查看 HTTP/2 帧信息。

你可以传给 `nghttp` 的一些参数列举如下：

- `-v`（打印 debug 信息）
- `-n`（丢弃下载的数据，如 HTML 内容）
- `-a`（下载在 HTML 中指明的、与 HTML 同一个域的引用资源）
- `-s`（打印统计信息）
- `-H <header>`（给请求添加首部，如 `-H':method: PUT'`）
- `--version`（打印版本信息，然后退出）

使用 `nghttp --help` 查看可用的参数列表。

下面的例子使用 nghttp 的 n 和 s 参数，忽略下载的数据，进行下载统计：

```
$ nghttp -ns https://akah2san.h2book.com/hello-world.html
***** Statistics *****

Request timing:
  responseEnd: the time when last byte of response was received
               relative to connectEnd
 requestStart: the time just before first byte of request was sent
               relative to connectEnd. If '*' is shown, this was
               pushed by server.
      process: responseEnd - requestStart
         code: HTTP status code
         size: number of bytes received as response body without inflation.
          URI: request URI

see http://www.w3.org/TR/resource-timing/#processing-model

sorted by 'complete'

id  responseEnd requestStart  process code size request path
  2   +142.85ms *   +35.89ms 106.96ms  200   64 /resources/push.css
 13   +175.46ms       +128us 175.33ms  200  169 /hello-world.html
```

仔细看这个结果，会发现由 HTTP/2 推送带来的一些有趣的东西。

- 尽管没有设置 a 参数，/resources/push.css 也加载出来了。这是因为该资源是由服务端推送的，即 requestStart 列前面的 * 所指示的。
- /resources/push.css 在 HTML **之前**就已经加载完成了。

5.7 节提供了完整的运行 nghttp 并检查输出结果的例子。

8.5 curl

curl[9] 最早是 Daniel Stenberg 所写的一个软件项目，它提供了在各种协议下传输数据的类库（libcurl）和命令行工具（curl）。尽管它使用 nghttp2 来做 HTTP/2 支持，但它的使用比 nghttp 更普遍，加上它具备 nghttp 所不具备的功能，也足以在你的调试工具箱中争得一席之地。截至本书写作时，curl 支持 261 个软件包以及 34 种操作系统。你可以使用 curl 下载指引 [10] 来找到合适的安装包。

使用curl

要使用 curl 模拟 HTTP/2 的请求，需要在命令行里传递 --http2 选项。添加 -v 参数将显示更多关于你访问的 URL 的调试信息。我们通过 openssl 可以获得的信息，大部分在 curl 的

注 9：https://github.com/curl/curl/

注 10：https://curl.haxx.se/dlwiz/

日志输出中更容易获得。来看下面这个例子：

```
$ curl -v --http2 https://akah2san.h2book.com/hello-world.html
*   Trying 2001:418:142b:19c::2a16...
* Connected to akah2san.h2book.com (2001:418:142b:19c::2a16) port 443 (#0)
* ALPN, offering h2                                              ❶
...
* SSL connection using TLSv1.2 / ECDHE-RSA-AES256-GCM-SHA384
* ALPN, server accepted to use h2
* Server certificate:                                            ❷
*  subject: CN=akah2san.h2book.com
*  start date: Aug 12 17:59:00 2016 GMT
*  expire date: Nov 10 17:59:00 2016 GMT
*  subjectAltName: host "akah2san.h2book.com" matched cert's
     "akah2san.h2book.com"
*  issuer: C=US; O=Let's Encrypt; CN=Let's Encrypt Authority X3
*  SSL certificate verify ok.
* Using HTTP2, server supports multi-use
* Connection state changed (HTTP/2 confirmed)
...
* Using Stream ID: 1 (easy handle 0x7f8d59003e00)                ❸
> GET /hello-world.html HTTP/1.1
> Host: akah2san.h2book.com
> User-Agent: curl/7.49.1
> Accept: */* >
>
* Connection state changed (MAX_CONCURRENT_STREAMS updated)!
* HTTP 1.0, assume close after body
< HTTP/2 200                                                     ❹
< server: Apache
< content-length: 169
...
<
<html>                                                           ❺
        <head lang="en">
                <meta http-equiv="Content-Type" content=
                  "text/html; charset=UTF-8">
                <title>Hello HTTP/2</title>
        </head>
        <body>Hello HTTP/2</body>
</html>
* Closing connection 0
* TLSv1.2 (OUT), TLS alert, Client hello (1):
```

❶ ALPN 信息。

❷ TLS 信息（跟我们在 `openssl` 中看到的类似）。

❸ 流数据。

❹ 我们使用 HTTP/2，并获得了一个 200 的响应。生活真美好。

❺ 页面内容。

测试网页加载时间

你可以使用 curl 的 `w` 参数打印出有价值的性能数据（参见 `man page`[11]）。

给 curl 请求添加以下这些参数（参数包含一些格式化文本的指令）：

```
-w "Connection time: %{time_connect}\t1st byte transfer:
  %{time_starttransfer}\tDownload time: %{time_total}
    (sec)\tDownload Speed: %{speed_download} (bps)\n"
```

你将看到以下字段：

- 连接时间
- 首字节时间
- 下载时间
- 总时间
- 下载速度（说明服务器每秒能发送多少字节）

例如：

```
$ curl -v --http2 https://akah2san.h2book.com/hello-world.html -w \
       "Connection time: %{time_connect}\t                          \
       1st byte transfer: %{time_starttransfer}\t                  \
       Download time: %{time_total} (sec)\t                        \
       Download Speed: %{speed_download} (bps)\n"

...omitting a bunch of lines...
* Connection #0 to host akah2san.h2book.com left intact
Connection time: 0.054 1st byte transfer: 0.166 Download time: 0.166 (sec)
Download Speed: 1016.000 (bps)
```

8.6 h2i

h2i[12] 是由 Brad Fitzpatrick 创建的交互式 HTTP/2 终端调试器，可以用来向服务器发送“未加工的”HTTP/2 帧。你可以用它跟 HTTP/2 服务器直接交互，跟之前用 `telnet` 或者 `openssl` 之类的工具与 h1 交互的方式非常像。

h2i 只需要指定支持 HTTP/2 协议的网站的域名即可。连接创建之后，你将看到一个 `h2i>` 提示符，你可以通过它向服务器发送 HTTP/2 帧。

下面的例子说明了如何用 h2i 发起客户端请求 https://www.google.com/（注意，较长的行使用 `<cut>` 截断了）：

注 11：https://curl.haxx.se/docs/manpage.html

注 12：https://github.com/bradfitz/http2/tree/master/h2i

```
$ h2i www.google.com
Connecting to www.google.com:443 ...
Connected to 172.217.5.100:443
Negotiated protocol "h2"
[FrameHeader SETTINGS len=18]
  [MAX_CONCURRENT_STREAMS = 100]
  [INITIAL_WINDOW_SIZE = 1048576]
  [MAX_HEADER_LIST_SIZE = 16384]
[FrameHeader WINDOW_UPDATE len=4]
  Window-Increment = 983041

h2i> headers
(as HTTP/1.1)> GET / HTTP/1.1
(as HTTP/1.1)> Host: www.google.com
(as HTTP/1.1)>
Opening Stream-ID 1:
 :authority = www.google.com
 :method = GET
 :path = /
 :scheme = https
[FrameHeader HEADERS flags=END_HEADERS stream=1 len=445]
  :status = "200"
  date = "Wed, 01 Mar 2017 00:08:06 GMT"
  expires = "-1"
  cache-control = "private, max-age=0"
  content-type = "text/html; charset=ISO-8859-1"
  p3p = "CP=\"This is not a P3P policy! See <cut>
  server = "gws"
  x-xss-protection = "1; mode=block"
  x-frame-options = "SAMEORIGIN"
  set-cookie = "NID=98=OOy2zBP3TY9GM37WXG9PFtN <cut>
  alt-svc = "quic=\":443\"; ma=2592000; v=\"35,34\""
  accept-ranges = "none"
  vary = "Accept-Encoding"
[FrameHeader DATA stream=1 len=16384]
  "<!doctype html><html itemscope=\"\" itemtype=\"http://schema.org/WebPage\"
   lang=\"en\"> <head><meta content=\"Search the world's information, including
   webpages, images, videos and more. Google has many special features to help
   <cut>
[FrameHeader PING len=8]
  Data = "\x00\x00\x00\x00\x00\x00\x00\x00"
h2i> quit
```

8.7 Wireshark

Wireshark[13] 是一款流行的网络包分析器，内置了数以百计的高层级协议的支持，其中也包含 HTTP/2。这说明它不仅可以像专业的工具（如 `tcpdump`）那样从网络中抓取数据包，还可以把数据包重组为你希望查看的高层级的协议。它不仅有 GUI 形式，还提供了命令行工具 `tshark`。

注 13：https://www.wireshark.org/

在 Wireshark 官网[14]有支持 Windows 和 macOS 的二进制程序可供下载安装。此外，还有大约 20 种 Unix/Linux 发行版下的安装包链接。Wireshark 支持良好，方便获取。

由于几乎所有的 HTTP/2 都是基于 TLS 的，使用 Wireshark 来查看 h2 变得复杂了。也就是说，在 Wireshark 导出的文件里，你可以看到 TLS 数据包，但是嗅探器不能解开数据包查看里面的内容。毕竟，这是 TLS 的设计宗旨。在 8.1.2 节（“Firefox 会话密钥日志”）和 8.1.1 节（“Chrome 会话密钥日志”），我们讨论了如何使用 Firefox 和 Chrome 记录密钥，以供 Wireshark 解开 TLS 数据包使用。依靠这个功能，加上当前跟随 Wireshark 一起发行的 HTTP/2 插件，可以清楚地看到在一个 HTTP/2 会话里都发生了些什么。

使用 `tshark` 命令，可以得到如下输出：

```
$ tshark port 443 and host www.example.com
Capturing on 'Wi-Fi'
  1   0.000000 TCP 78 65277→443 [SYN] Seq=0 Win=65535 Len=0 MSS=1460
       WS=32 TSval=1610776917 TSecr=0 SACK_PERM=1
  2   0.096399 TCP 74 443→65277 [SYN, ACK] Seq=0 Ack=1 Win=14480 Len=0 MSS=1460
       SACK_PERM=1 TSval=2815107851 TSecr=1610776917 WS=128
  3   0.096489 TCP 66 65277→443 [ACK] Seq=1 Ack=1 Win=131744 Len=0
       TSval=1610777007 TSecr=2815107851
  4   0.096696 SSL 264 Client Hello
...
 33   0.386841 TCP 66 65277→443 [ACK] Seq=1043 Ack=7845 Win=128160
       Len=0 TSval=1610777288 TSecr=2815108131
 34   0.386842 TCP 66 [TCP Window Update] 65277→443 [ACK] Seq=1043
       Ack=7845 Win=131072 Len=0 TSval=1610777288 TSecr=2815108131
 35   0.386887 TCP 66 65277→443 [ACK] Seq=1043 Ack=9126 Win=129760
       Len=0 TSval=1610777288 TSecr=2815108131
 36   0.436502 HTTP2 143 HEADERS
 37   0.535887 TCP 1514 [TCP segment of a reassembled PDU]
 38   0.536800 HTTP2 1024 HEADERS, DATA
 39   0.536868 TCP 66 65277→443 [ACK] Seq=1120 Ack=11532
       Win=130112 Len=0 TSval=1610777433 TSecr=2815108271
```

这个示例日志使你可以深入到 TCP、TLS 和 HTTP/2 的内部。其他的选项[15]允许你深入挖掘这些协议，准确地看到正在发生的事情。

8.8 小结

你可以使用本章介绍的工具来完成一些轻量任务，比如验证你现有的证书是否包含必需的 h2 加密因子，HTTP 通信的底层调试，或者构建简单的 h2 Web 服务器来做更高级的测试。知道如何完成这类任务，有助于深入了解 h2，并帮你将网站升级到支持 h2 协议。

注 14：https://www.wireshark.org/download.html

注 15：https://www.wireshark.org/docs/man-pages/tshark.html

第 9 章

展望未来

HTTP/1.1 与 HTTP/2 相隔了大概 20 年，但是从目前的研究和实验情况来看，我们不可能等上几十年才升级到下一个版本。h2 在逐渐普及，但仍有一些工作在持续展开，它们极有可能在互联网协议世界产生新的重大变革。为了理解这种快速演变的源头及其背后的原因，我们有必要探究 HTTP 技术栈背后的世界。

9.1 TCP还是UDP

如今你仍然会听到很多这方面的讨论，虽然这的确是个问题，但有些讨论也有误导性。在第 3 章分析 h2 的设计初衷时，我们曾经讨论过 TCP。这里快速回顾一下，TCP 是基于 IP 报文的协议，它包含了一些众所周知的概念：连接、可靠性、拥塞控制。这些因素能够保证 TCP 的可靠性，也能保证它在繁忙的网络中持续工作。相比之下，UDP（用户数据报协议）就显得更加原始。在 UDP 下，数据报（包）是相互独立的，包与包之间没有什么关系。它没有所谓的“连接”原语，没有传输可靠性保证，也无法适应不同的网络条件；因此，UDP 非常适合像 DNS 这样简单的应用协议，它只包含小型独立查询请求和响应。

如果要让基于 UDP 的应用能够成为下一代 Web 所构建的基础，我们需要实现连接原语，引入可靠性保证，并提供或多或少的拥塞控制。换句话说，我们需要重复实现 TCP 已经提供了的很多特性。所以，如果在数据包级别上这两个协议几乎相同，只是单纯为了让 UDP 对浏览器有用，我们就需要重新实现 TCP 技术栈的很多（如果不是全部）功能。那么，从 TCP 转向 UDP 的初衷是为了什么？为什么不干脆修改 TCP，然后接着用呢？

答案与 TCP 实现的方式有关。大多数现代操作系统在内核中直接提供了 TCP 协议栈实现。

很久以前就是如此了，这是出于性能的考虑。不管怎么说，修改内核比较麻烦。这可不是未来某个高质量浏览器的开发人员能够独自完成的。内核变更往往来自操作系统开发商，只有操作系统更新后，它才会生效。因为相比于浏览器更新而言，操作系统更新代价一般更高些，所以操作系统变更的成本会让我们少做变更，并且两次变更之间的间隔更长。设想一下，为了让变更生效，互联网上的很多基础设施也必须同步更新。虽然修订 TCP 是可能的，但是这和“现实可行”不沾边。

那么，为什么有人会想要基于 UDP 在用户空间重新实现 TCP 协议栈呢？简要回答是为了获得控制权。把 TCP 协议栈移到用户空间——例如，包含于浏览器自身 ——就能将开发人员对网络协议栈的控制权提升到前所未有的高度。他们还能尽快开发、部署、迭代新版本，因为用户只要自动更新浏览器就可以了。这是尝试者的梦想，也是 TCP 与 UDP 之争的症结所在。

因此，问题不是“TCP 还是 UDP”，而是“内核空间还是用户空间”。如果下次听到有人大肆赞扬某个协议的差异特性，请记住这一点。

9.2 QUIC

现在不必考虑 TCP 与 UDP 之争了，接下来开始真正讨论一些前瞻性技术，其中有些甚至已经投入使用。

HTTP/2 的弱点之一就是依赖主流 TCP 实现。在 3.1.3 节中已经讨论过，TCP 连接受制于 TCP 慢启动、拥塞规避，以及不合理的丢包处理机制。用单个链接承载页面涉及的所有资源请求，就能享受多路复用带来的好处；然而面对 TCP 层级的队首阻塞时，我们还是束手无策。这并不影响单 TCP 链接成为杰作，不过任何杰作都有继续提升的空间。

Google 开发的 QUIC 是快速 UDP 网络连接（Quick UDP Internet Connection）的首字母缩写。QUIC 采纳了 HTTP/2 的优点，构建在驻留于用户空间的、基于 UDP 的传输协议之上（参见 9.1 节），并加入加解密、身份认证以及其他特性，可以说简直就是萃取了协议的精华。下面一段文字摘自它的 RFC 草案版本 [1]：

> QUIC 提供 HTTP/2 等效的多路复用与流控、TLS 等效的安全机制，以及 TCP 等效的连接语义、可靠性、拥塞控制。

这真是令人印象深刻。HTTP/2 为了妥善应对它所依赖的 TCP 而颇费周折；QUIC 则抛开了那些限制，转向精益求精。牛顿曾在 1676 年描述过这种进步，他说：

> 如果我比别人看得更远，那是因为我站在巨人的肩上。
>
> ——艾萨克·牛顿

注 1：https://tools.ietf.org/html/draft-hamilton-early-deployment-quic-00

QUIC 包含如下重要特性，它们能填补 h2 留下的空白。

无序的包处理

如果 TCP 流上丢了一个数据包，那么整个 h2 连接都会停顿下来，直到该数据包重发并被接收到。QUIC 将允许应用层继续接收并处理那些来自未受到丢包影响的流的包。

灵活的拥塞控制

QUIC 的拥塞控制机制被设计成插件式的，所以它非常容易实验新的算法，甚至可以根据实时条件切换不同算法。

更低的建立连接开销

QUIC 的目标是建立连接时的“零”往返时延（0-RTT），包括加解密以及身份认证。依据当今的技术（TCP 和 TLS 1.2），建立连接所需的最小往返数是 3。

传输细节的身份验证

在注入攻击以及针对 TCP 天生的信任特点开展的攻击手段面前，目前的 TCP 是比较脆弱的。QUIC 会验证包的首部，增加对这类攻击的防御程度（虽然不能完全避免）。

连接迁移

移动互联网时代，在“逻辑”长连接的通信当中，IP 地址可能会变化。从 TCP 的角度看，连接需要断开再重连。即使客户端在移动，QUIC 也会尝试提供维持连接的语义。

虽然 RFC 还没有完成，现在 QUIC 已经有一个实现版本，可以在 Chrome 和许多 Google 公司的其他产品中使用，你可以立刻体验。

9.3 TLS 1.3

TLS（传输层安全协议）是 HTTP/2 所需的加解密以及身份认证层。虽然我们似乎刚到 TLS 1.2，其实相关 RFC 已经发布近 10 年了。[2]TLS 1.3 目前还处于开发中，截至 2017 年 3 月，已经进展到 RFC 的草案 19。[3] 它是对 TLS 1.2 的重要整理；除了支撑已有协议，更重要的是，对于我们的讨论而言，它还实现了若干增强性能的特性。

TLS 1.3 中最明显的改进提议是，新连接只需要 1 次往返时延（目前最少需要 3 次）；如果是恢复连接的话，不需要往返时延（0-RTT）。这种趋势可以总结为“让我们消灭万恶的往返时延”。

注 2：https://www.ietf.org/rfc/rfc5246.txt

注 3：https://tools.ietf.org/html/draft-ietf-tls-tls13-19

9.4　关于HTTP/3

会不会有 HTTP/3？如果有，它会是什么样？

第一个问题的答案无疑是“当然会”。我们所处的年代，Web 协议在迅速实验，也在迅速实现。速度至关重要，并且对网站获取以及留存用户有直接影响，继而会直接影响网站完成目标，这些目标可能是收获利润、信息发布，或者建立人与人之间的连接。另外，协议实现者、网络运营者以及其他人越来越意识到，虽然互联网是不分国界的，但是网络接入和连接质量却有很大差异。并不是所有人都拥有低延迟、高带宽的连接。夸张点说，性能非常重要，我们将继续努力推动通信能力，直到接近物理极限。

至于 HTTP/3 会是什么样，这个问题更加有趣。HTTP/2 是受了 SPDY 相关概念的启发，并且使用 SPDY 作为第一份草案的起点。HTTP/3 是否会利用 QUIC 做相似的事情，是否会有各方面更加完美的新生协议出现？现在还不知道，但是大致可以这么说，任何集成进 h3 的东西都将更快速、更可靠、更安全，并在更多变的网络连接状况下提供更多弹性。

9.5　小结

HTTP/2 是新生事物。这意味着有更多有趣的东西亟待发现。我们花了近 20 年时间优化 HTTP/1——事实上，围绕它产生了整个行业。将被整合进 HTTP/3 里的内容，有些可能已经众所周知，但是将来也许会有更棒的想法。离开 h1 并从 h2 开始，把边界往外扩展，打破常规，然后思考和学习——如果你已经这么做了，请分享出来。HTTP/3 就应当这样炼成。

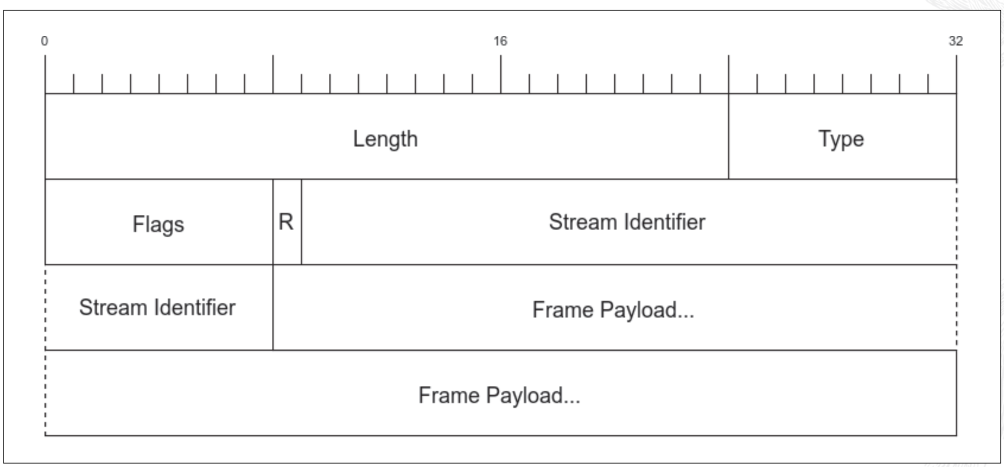
附录A

HTTP/2帧

本附录是 HTTP/2 分帧层的一份简要参考。各节分别讲解帧类型数字、帧的二进制数据格式、帧的描述，以及对应帧的标识位列表。

A.1　帧首部

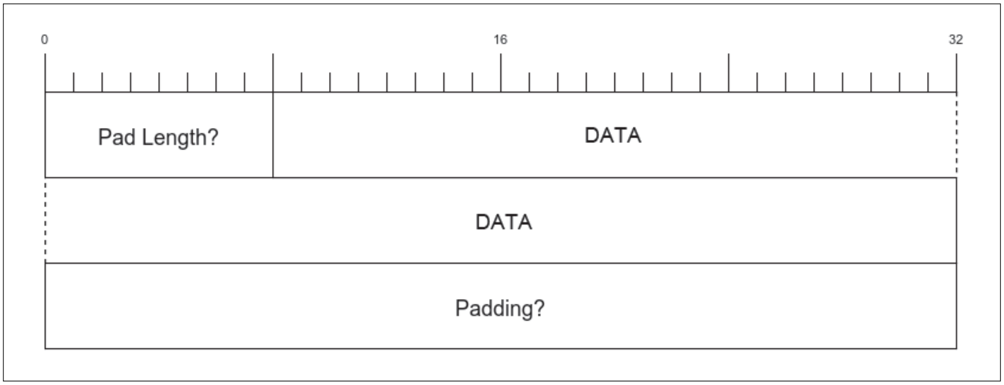
如第 5 章所述，每个帧以相同的 9 个字节开始。



各字段的描述参见表 5-1。

A.2 DATA帧

DATA 类型的帧包含的字节长度不定。换言之，这些帧包含了请求和发送的对象。如果超出帧容许的最大长度，资源数据会被切分到一个或者多个帧里面去。在某些情况下，还会包含填充长度（Pad Length）字段和填充数据（Padding），以隐藏真实的消息大小（出于安全方面的考虑）。



A.2.1 DATA帧字段

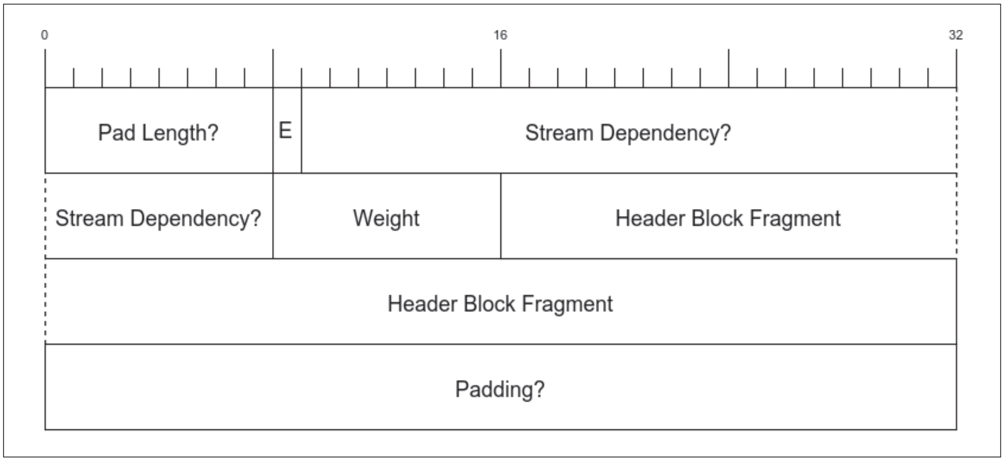
名称	长度	描述
Pad Length（填充长度）	1 字节	填充字节的长度；在帧首部的 `PADDED` 标识设置为 1 的时候才会有该字段
DATA（数据）	长度可变	帧的内容
Padding（填充数据）	长度可变	长度为 Pad Length 字段的值，所有的字节被设置为 0

A.2.2 DATA帧标识位

名称	位	描述
END_STREAM	0x1	表明这是流中最后的帧（流终止）
PADDED	0x8	表明此帧添加了填充数据，要处理 Pad Length 和 Padding 字段

A.3 HEADERS帧

HEADER 帧用以创建流，并向另一端发送消息首部。



A.3.1 HEADERS帧字段

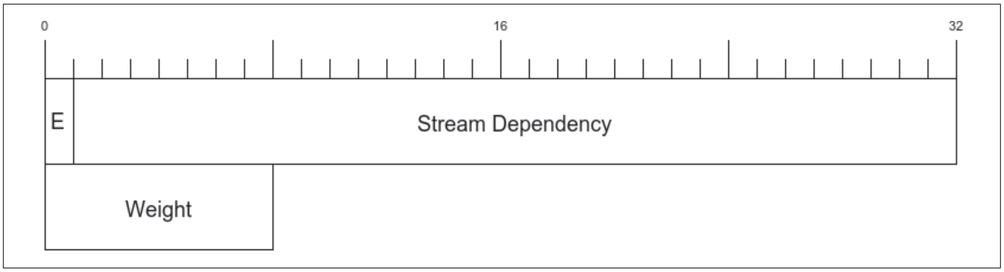
名称	长度	描述
Pad Length（填充长度）	1 字节	填充字节的长度；帧首部的 PADDED 标识设置为 1 时才会有该字段
E	1 位	表示流依赖是否为专用的；只有设置了 PRIORITY 标识才会有该字段
Stream Dependency（流依赖）	31 位	表示当前流所依赖的流，如果有的话；只有设置了 PRIORITY 标识才会有该字段
Weight（权重）	1 字节	当前流的相对权重；只有设置了 PRIORITY 标识才会有该字段
Header Block Fragment（首部块片段）	长度可变	消息的首部
Padding（填充数据）	长度可变	长度为 Pad Length 字段的值，所有的字节被设置为 0

A.3.2 HEADERS帧标识位

名称	位	描述
END_STREAM	0x1	表明这是流中最后的帧（流终止）
END_HEADERS	0x4	表明这是流中最后一个 HEADERS 帧；如果此标识未设置，表示随后会有 CONTINUATION 帧
PADDED	0x8	表明此帧添加了填充数据，要使用 Pad Length 和 Padding 字段
PRIORITY	0x20	如果设置了此标识，就表示要使用 E、Stream Dependency 以及 Weight 字段

A.4 PRIORITY帧

发送 PRIORITY 帧是为了标识流的优先级。它可以多次发送，后面指定的优先级会覆盖之前的。



PRIORITY帧字段

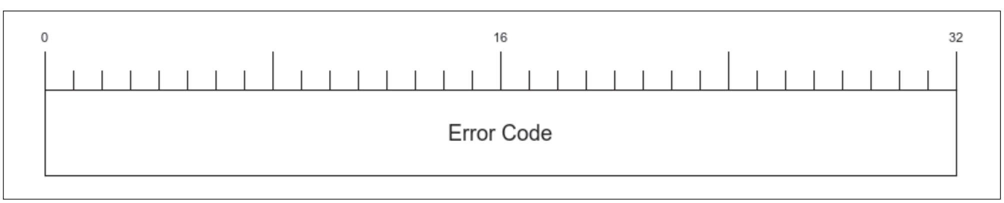
名称	长度	描述
E	1 位	标识当前的流是否为专用，是否不依赖其他流
Stream Dependency（流依赖）	31 位	如果当前流依赖其他流，标识其所依赖的流
Weight（权重）	1 字节	当前流的相对权重

PRIORITY 帧没有专用标识。

A.5　RST_STREAM帧

如果要终止一个流，可以将 RST_STREAM 加在该流的两端。这通常是为了处理某种错误。

帧里的 Error Code（错误码）字段用来标注重置的原因。关于错误码的列表，可以参考 RFC 7540 的第 7 节 [1]。

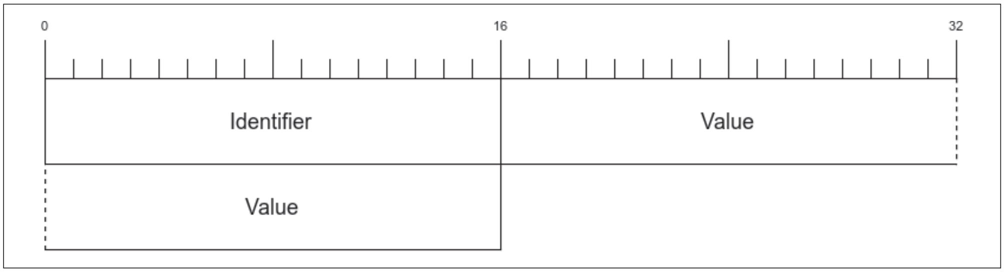


A.6　SETTINGS帧

SETTINGS 帧包含了若干有序的键 / 值对 [2]。键 / 值对的数量等于帧长度除以单组设置的长度（共 6 字节，Identifier 的 2 字节加上 Value 的 4 字节）。

注 1：https://tools.ietf.org/html/rfc7540#section-7

注 2：键指下图中的 Identifier。——译者注



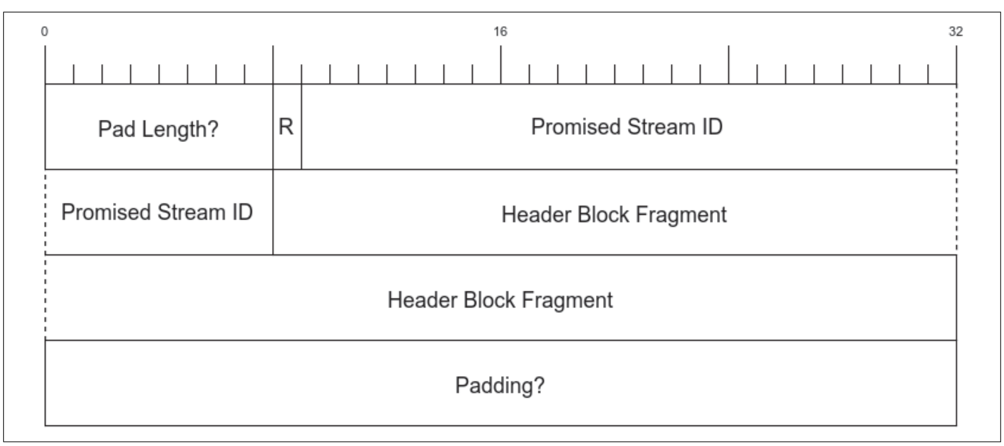
SETTINGS帧参数列表

名称	ID	默认值	描述
SETTINGS_HEADER_TABLE_SIZE	0x1	4096	重新指定 HPACK 所用的首部表的最大尺寸
SETTINGS_ENABLE_PUSH	0x2	1	如果设置为 0，当前端不会发送 PUSH_PROMISE 帧
SETTINGS_MAX_CONCURRENT_STREAMS	0x3	无限制	表明发送端能够并行接收的流的最大数量
SETTINGS_INITIAL_WINDOW_SIZE	0x4	65353	表明发送端流量控制的初始窗口尺寸
SETTINGS_MAX_FRAME_SIZE	0x5	16384	发送端希望接收的最大帧尺寸；这个值必须介于初始值和 16 777 215（$2^{24}-1$）之间
SETTINGS_MAX_HEADER_LIST_SIZE	0x6	无限制	该设置告诉通信的另一端，本端期望接收的最大首部的尺寸

如果一端接收并处理了 SETTINGS 帧，就必须返回一个 SETTINGS 帧，在帧首部中带上 ACK 标识（0x1）。这是 SETTINGS 帧里定义的唯一的标识位。这样发送端就知道接收端收到了新的 SETTINGS 帧，并会遵守 SETTINGS 帧的设置。

A.7　PUSH_PROMISE帧

服务端发送 PUSH_PROMISE 帧来告诉客户端，它将发送一份客户端尚未明确请求的资源。PUSH_PROMISE 帧实际上是对客户端发送的 HEADERS 帧的补充。



A.7.1　PUSH_PROMISE帧字段

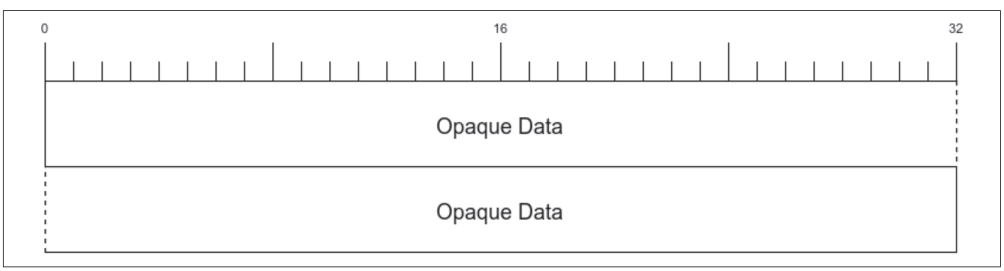
名称	长度	描述
Pad Length（填充长度）	1 字节	填充字节的长度；在帧首部的 PADDED 标识设置为 1 时才会有该字段
R	1 位	保留位，不必设置
Promised Stream ID（预期的流 ID）	31 位	告知发送端将要使用的流 ID（总是偶数，因为是由服务端发送的）
Header Block Fragment（首部块片段）	长度可变	推送的消息首部
Padding（填充数据）	长度可变	长度为 Pad Length 字段指定的值，各字节均为 0

A.7.2　PUSH_PROMISE帧标识

名称	位	描述
END_HEADERS	0x4	表明这是流的最后一个 HEADERS 帧；如果此标识未设置，说明随后会有 CONTINUATION 帧
PADDED	0x8	说明此帧包含了填充数据，要设置 Pad Length 和 Padding 字段

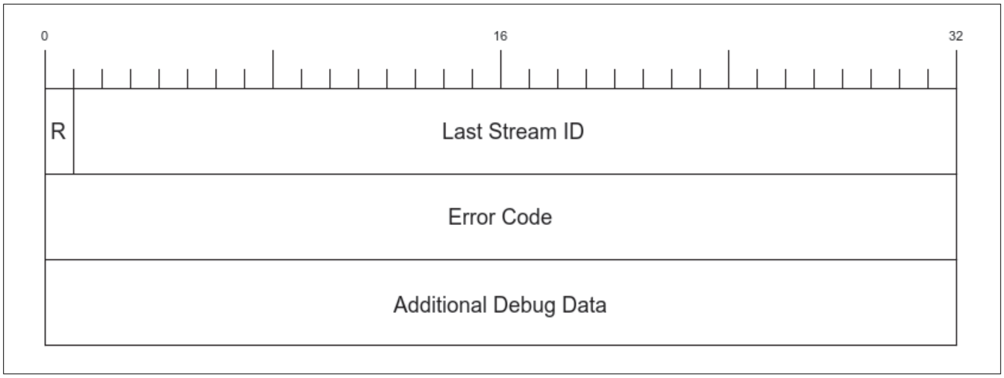
A.8　PING帧

PING 帧用以计算两端之间的往返时间。此帧有一个标识位 ACK（0x1）。如果一端收到一个不带 ACK 的 PING 帧，它就必须返回一个 PING 帧，这个帧必须设置 ACK 标识，并且包含同样的数据内容（Opaque Data）。需要注意的是，PING 帧不属于任何一个流（它们是连接层的），因此它们的流 ID 要设置为 0x0。



A.9　GOAWAY帧

GOAWAY 帧用以礼貌地关闭连接。这是连接层的帧，并且发送时流 ID 要设置为 0x0。通过发送 GOAWAY 帧，当前端可以清晰地告诉接收端，它接收到了什么、未接收到什么，以及什么原因（如果有的话）导致了 GOAWAY。如果出了问题，错误码将被设置为 RFC 7540 第 7 节中定义的某个错误码，并且 Last Stream ID（最后一个流 ID）会被设置为曾经处理过的最大的流 ID。如果没有错误发生，而当前端要断开连接（浏览器标签页关闭以及连接超时等情况），那就发送 NO_ERROR（0x0）这个错误码，并且 Last Stream ID 被设置为 $2^{31}-1$。

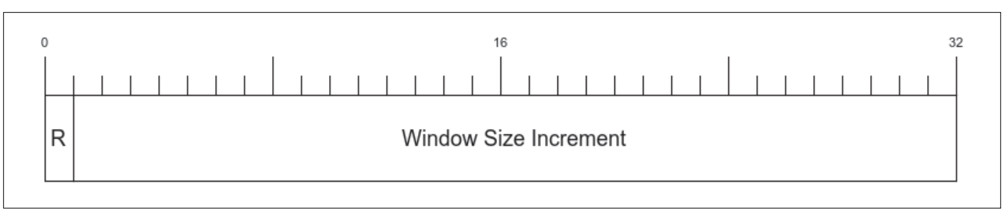


GOAWAY帧字段

名称	长度	描述
R	1 位	保留位
Last Stream ID（最后的流 ID）	31 位	GOAWAY 的发送端接收 / 处理的最大的流 ID；发送这个值之后，接收方可以清楚地知道发送方接收到了什么，以及没有接收到什么
Error Code	4 字节	h2 定义的错误码，或者成功关闭时的 `NO_ERROR` 码
Additional Debug Data	长度可变	发送方可能发送的其他数据内容，说明当前的状态或者其他问题

A.10　WINDOW_UPDATE帧

WINDOW_UPDATE 帧用来做流量控制——发送方发送一个 WINDOW_UPDATE 帧，告诉接收方自己此时期望接收多少字节。流量控制可以应用到单个的流，也可以应用到连接承载的所有流（流 ID 为 0x0）。需要注意的是，在单个流上指定的 WINDOW_UPDATE 帧也会作用于连接层的流量控制。



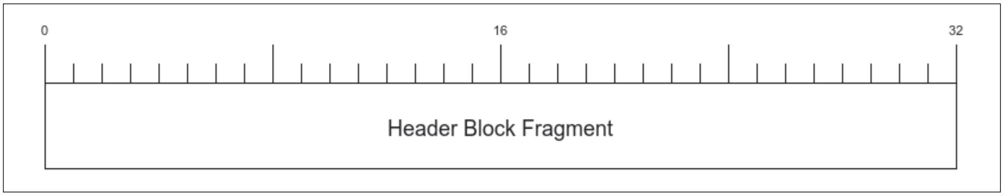
WINDOW_UPDATE帧字段

名称	长度	描述
R	1 位	保留位
Window Size Increment（窗口大小增量）	31 位	当前窗口可以增加的字节数

WINDOW_UPDATE 帧没有专用标识。

A.11 CONTINUATION帧

CONTINUATION 帧包含之前的 HEADERS、PUSH_PROMISE，或者 CONTINUATION 帧的附加首部。



A.11.1 CONTINUATION帧字段

名称	长度	描述
Header Block Fragment（首部块片段）	长度可变	具体描述参见 HEADERS 帧

A.11.2 CONTINUATION帧标识位

名称	位	描述
END_HEADERS	0x4	表明这是流中最后的 HEADERS 帧；如果此标识没有设置，说明后面还有 CONTINUATION 帧

附录B

工具引用

本书涉及的所有工具都可以在互联网上找到，但为了方便查找并保证本书的完整性，同时也为了以某种形式标注本书的写作时间，下面列出了本书用到的所有工具。

B.1 工具

应用	URL
openssl	https://www.openssl.org/
nghttp2	https://nghttp2.org/
cURL	https://curl.haxx.se/
h2i	https://github.com/bradfitz/http2/tree/master/h2i
Wireshark	https://www.wireshark.org/
Certbot	https://certbot.eff.org/

B.2 浏览器

应用	URL
微软 Edge	https://www.microsoft.com/en-us/windows/microsoft-edge
Chrome	https://www.google.com/chrome/
Firefox	https://www.mozilla.org/en-US/firefox
Safari	http://www.apple.com/safari/
Opera	https://www.opera.com/

B.3　服务器、代理和缓存

应用	URL
h2o	https://h2o.examp1e.net/
Apache	https://httpd.apache.org/
Squid	http://www.squid-cache.org/
IIS	https://www.iis.net/
nginx	https://www.nginx.com/
varnish	https://varnish-cache.org/
Jetty	http://www.eclipse.org/jetty/
Caddy	https://caddyserver.com/
Apache Traffic Server	https://trafficserver.apache.org/

关于作者

Stephen Ludin，Akamai 公司 Web 性能部门首席架构师。他带领 Akamai 公司的 Foundry 团队，负责研发下一代 Web 技术。他同时也是互联网安全研究小组（Let’s Encrypt 项目的母机构）以及 Rubicon Labs 的成员。

Ludin 毕业于加州大学圣迭戈分校电子音乐制作专业；就读期间，他用 C 语言程序创作实验音乐。后来，他决定利用在音乐世界中获得的充沛的创造力、技术能力、管理能力，力图让 Web 对电子商务与信息沟通而言更快捷、更安全。

Javier Garza，多门编程语言的技术布道者。他喜欢拆东西、了解内部原理，并找到改进的最佳实践。他 9 岁就开始破解基于 BASIC 的计算机游戏；在过去 25 年里，他一直与计算机打交道，先后工作于西班牙、德国和美国（在美国的半数时间供职于 Akamai，帮助互联网上一些最大的网站更快、更安全地运行）。

关于封面

本书封面的动物是金背地松鼠（学名 Callospermophilus lateralis），属于松鼠科，广泛分布于北美西部地区，可以在森林、草地和干旱的平原环境中生存。

由于外观相似，金背地松鼠很容易和花栗鼠混淆。两者都有黑色条纹，从背部往下延伸，但那些条纹没有延伸到金背地松鼠的脸部。金背地松鼠体型上也更大一些（9~12 英寸长），重量约 113~396 克。金背地松鼠的尾巴比花栗鼠要长，眼周有白色的圆圈。它们得名自头顶上金棕色的皮毛。

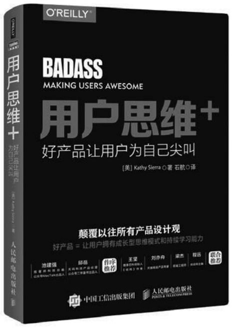
金背地松鼠是杂食性动物，将收集到的食物存储在颊囊中。它们要储存足够的脂肪才能熬过冬眠。它们的食物包括种子、水果、昆虫、菌类、鸟蛋，甚至还有一些小型脊椎动物。冬眠时它们会留下一些食物，等到春暖花开，它们从冬眠的洞里醒来时再吃。

金背地松鼠的繁殖期通常是在春季。雄性比雌性更早从冬眠中苏醒过来，繁殖要等到双方都苏醒之后才开始。妊娠持续约 28 天，雌性松鼠会生下大约 5 只无毛幼仔。幼仔将在出生 3~6 周后断奶并开始独立生活。

O’Reilly 封面上的许多动物都濒临灭绝，它们都是这个世界的至宝。想要了解如何提供帮助，请访问 animals.oreilly.com。

本书封面图片来自 *Pictorial Museum of Animated Nature*。

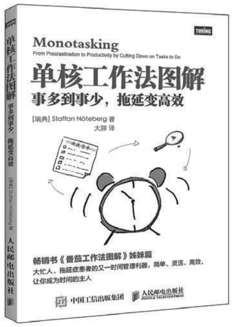
技术改变世界 · 阅读塑造人生



用户思维：好产品让用户为自己尖叫

- ◆ 颠覆以往所有产品设计观
- ◆ 好产品 = 让用户拥有成长型思维模式和持续学习能力
- ◆ 极客邦科技总裁池建强、公众号二爷鉴书出品人邱岳作序推荐
- ◆ 《结网》作者王坚、《谷歌和亚马逊如何做产品》译者刘亦舟、前端工程师梁杰、优设网主编程远联合推荐

书号： 书号：978-7-115-45742-4
定价： 69.00 元



单核工作法图解

- ◆ 畅销书《番茄工作法图解》姊妹篇
- ◆ 大忙人、拖延症患者的又一时间管理利器，简单、灵活、高效，让你成为时间的主人
- ◆ 吴晓波、战隼、高地清风、采铜、叶骥联合力荐
- ◆ 随书赠送精美海报、书签

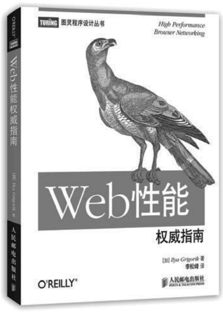
书号： 978-7-115-44860-6
定价： 39.00 元



用数据讲故事

- ◆ 学会用数据讲故事，让沟通更高效、更直接
- ◆ 告别粗糙图表和PPT，让客户满意，给自己加分
- ◆ 前Google人力分析团队经理作品//盖茨基金会、摩根大通银行等组织高管鼎力推荐

书号： 978-7-115-46011-0
定价： 59.00 元



Web 性能权威指南

- ◆ Web性能优化上乘之作， IETF下一代HTTP协议（HTTP /2）工作组主席力荐

书号： 978-7-115-34910-1
定价： 69.00 元

O'REILLY®

HTTP/2基础教程

让网站和应用更快速、更简洁、更稳健，从而有效提升用户体验，这是众多开发者梦寐以求的。然而互联网发展日新月异，HTTP/1.1协议已经难以满足现今的需求。在众多Web性能提升方案中，HTTP/2无疑值得尝试。

本书是HTTP/2实用指南，介绍了HTTP/2的设计初衷和新特性，以及如何才能充分利用这些特性来打造高性能网站及应用。作者用定量分析方法，对比了不同网络环境下及不同浏览器上HTTP/1.1与HTTP/2的性能差异，并指出了网站迁移到HTTP/2需要注意的问题及对策。

本书主要内容：

- **HTTP发展回顾——面临性能挑战，促使协议升级**
- **HTTP/2概览——优点及迁移方法**
- **既有的建议方案，以及提升Web性能的技巧**
- **HTTP/2支持的浏览器、服务器、代理，以及内容分发网络**
- **相比于HTTP/1.1，采用HTTP/2的网站在性能上有何提升**
- **HTTP/2对网络通信中一些具体问题的改进，如延迟、丢包、首字节时间等**

Stephen Ludin，Akamai公司Web性能部门CAO，带领Akamai公司的Foundry团队，负责研发下一代Web技术。

Javier Garza，Akamai公司高级企业架构师，专注于维护公司的主要客户及战略合作伙伴。他也是Web性能优化和HTTP/2的布道者，为Akamai的产品开发团队提供技术建议和客户反馈。

“我强烈推荐使用HTTP/2，因为像这样整体提升网站速度的、服务器级的Web性能优化方案，实在是屈指可数。”

——Steve Souders
SpeedCurve联合创始人，Web性能优化先驱

“本书覆盖了HTTP/2中有价值的各种新特性，对很多技术点做了深入讲解和探讨，堪称目前HTTP/2相关图书中的佼佼者。”

——余晟
沪江教育集团技术中心研发总监

“距离HTTP/2正式发布已经两年多时间了，但相关图书文档都比较缺乏。这本《HTTP/2基础教程》虽然不厚，但满满的都是干货，值得所有软件开发者拥有。”

——李锟
Web架构师，Web开发老兵

封面设计：Karen Montgomery 张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机 / 网络技术 / HTTP

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-115-47389-9

ISBN 978-7-115-47389-9
定价：49.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks